AFRL-RI-RS-TR-2013-067

MACHINE INTELLIGENCE

MARCH 2013

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2013-067 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /

ALEX F. SISTI, Chief
Advanced Planning and Autonomous C2 Systems Branch

/ S /

JULIE BRICHACEK, Chief
Information Systems Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MARCH 2013	FINAL TECHNICAL REPORT	FEB 2010 – SEP 2012

4. TITLE AND SUBTITLE

MACHINE INTELLIGENCE

5a. CONTRACT NUMBER
IN-HOUSE

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62788F

6. AUTHOR(S)

Steven Loscalzo, Nathaniel Gemelli, Robert Wright

5d. PROJECT NUMBER
S2TS

5e. TASK NUMBER
IH

5f. WORK UNIT NUMBER
ML

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/Information Directorate
Rome Research Site/RISC
525 Brooks Road
Rome NY 13441-4505

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/Information Directorate
Rome Research Site/RISC
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2013-067

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2013-1080
Date Cleared: 7 MAR 2013

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The objective of this effort was to produce technologies that could reduce manpower requirements and improve response times of future command and control (C2) systems through research in machine learning. This in-house research effort explored various areas of machine learning to produce technologies capable of supporting future C2 systems. Machine learning has the potential to reduce manpower requirements, reduce decision cycle times, and improve the robustness of C2 systems. However, many obstacles, such as intractability in large state spaces, prevent the application of these technologies to practical C2 problems. The goal, then, was to research and develop innovative technologies that overcome said obstacles and enable the application of machine learning to relevant C2 problems. This goal was achieved through the research and development of new state space abstraction and feature selection algorithms. Theoretical and empirical results of this effort were published to refereed conferences and showcased in technology demonstrations. In this document, we detail our approaches and report our results in improved scaling of reinforcement learning via feature set reduction and state space abstraction.

15. SUBJECT TERMS
Machine Learning, Reinforcement Learning, Feature Set Reduction, State Space Abstraction

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**NATHANIEL GEMELLI**
U	U	U	UU	51	19b. TELEPONE NUMBER (*Include area code*) **N/A**

Contents

List of Tables

List of Figures

1 SUMMARY

The objective of this effort was to produce technologies that could reduce manpower requirements and improve response times of future command and control (C2) systems through research in machine learning. This in-house research effort explored various areas of machine learning to produce technologies capable of supporting future C2 systems. Machine learning has the potential to reduce manpower requirements, reduce decision cycle times, and improve the robustness of C2 systems. However, many obstacles, such as intractability in large state spaces, prevent the application of these technologies to practical C2 problems. The goal, then, was to research and develop innovative technologies that overcome said obstacles and enable the application of machine learning to relevant C2 problems. This goal was achieved through the research and development of new state space abstraction and feature selection algorithms. Theoretical and empirical results of this effort were published to refereed conferences and showcased in technology demonstrations. In this document, we detail our approaches and report our results in improved scaling of reinforcement learning through feature set reduction and the development of cutting edge state space abstraction methods.

2 INTRODUCTION

Reinforcement learning (RL) is designed to learn optimal control policies from unsupervised interactions with the environment. Many successful RL algorithms have been developed, however, none of them can efficiently tackle problems with high-dimensional state spaces due to the "curse of dimensionality," and so their applicability to real-world scenarios is limited. In this report, we detail our approaches to dealing with the high-dimensionality state space issues in five key areas: Evolutionary Tile Coding, Continuous State Space Abstraction, Density-based Automatic State Space Abstraction, Incremental Feature Selection and Sample Aware Feature Selection. These approaches are based around the idea of reducing the actual state space that is learned by abstracting the states in the space to a higher level, thus reducing the memory and search overhead, as well as reducing the actual number of features that are being considered from the set of possible perceptions available.

The remainder of this document is structured as follows. Section 3 will describe the algorithms developed in each of the key five areas mentioned above, give background on the particular problem the approach is addressing and discuss the other relevant work in that area. In Section 4 we will provide experimental results pertaining to each of the five algorithms along with discussion about the relevance of our results. Finally, in Section 5 we will provide concluding remarks.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

Reinforcement learning algorithms are often tasked with learning an optimal control policy in environments with complex state spaces. Since it is infeasible to learn the optimal action to take for every possible observation in most real-world environments with complex state spaces, techniques that can simplify such spaces must be utilized for learning to be effective and efficient. The two general techniques we have studied and applied in this effort are *state space abstraction* and *feature selection*.

State space abstraction techniques generalize the space into very few abstract states. These techniques must take care to avoid creating abstractions that prevents learning the optimal policy. Many commonly used abstractions, such as CMAC [1], can take considerable effort to tune to ensure

a learnable abstraction is created. Feature selection techniques are used to selectively discover or determine a subset of the complete set of features sufficient to solve a given learning problem. Ideally, the subset of features selected will be the minimal set necessary to represent the problem so that optimal policies can be found.

In this section, we detail the state space abstraction and feature selection techniques developed. The following section will show empirical results.

3.1 Evolutionary Tile Coding

Tile coding is a form of state abstraction for domains with continuous states spaces. It discretizes the state space into *tiles* that cover ranges of values for each feature in the state space. Every state that falls under a specific tile is treated as the same abstract state and the RL algorithm learns over the abstract state space. The effectiveness of tile coding methods depends heavily on the design of the tiling scheme. If there is insufficient resolution in a particular area of the state space the RL algorithm will not be able to find π^*. As a result the design and implementation of tile coding schemes has been a manual and time consuming process that requires significant domain expertise to be effective.

Recently, there has been work in automated tiling methods that attempt to derive an effective tiling scheme on-line [2, 3]. In this paper we introduce a new automated tile coding algorithm called Evolutionary Tile Coding (EvoTC). EvoTC uses a genetic algorithm to derive efficient tile structures that maximizes an RL algorithm's ability to find a good policy. We compare the performance of EvoTC to competing fixed and automated tile coding methods, CMAC and Adaptive Tile Coding. And we show that EvoTC is able to provide more efficient tile based state abstractions that should help enable RL algorithms to scale towards more complex problems.

The rest of this section proceeds as follows. First we provide background and details on the two tile coding approaches we use for comparison. We then introduce and describe EvoTC in detail. This is followed by a description of our experimental setup and results. We conclude with a discussion of the results and a summary of the conclusions we were able to make.

3.1.1 Background

Cerebellar Model Articulation Controller The Cerebellar Model Articulation Controller algorithm, better known as CMAC, was introduced in [4] (then called the Cerebellar Model Arithmetic Computer) as a means of providing local generalization of the state space based on how the human brain is thought to respond to stimuli [1]. This behavior allows states that are in proximity to an observed state to learn even though those states have not been observed themselves. It was chosen for our analysis because it is arguably the most popular of the tile coding methods [5].

CMAC partitions state spaces into a fixed set of non-overlapping tiles. Q-values that are learned from any one state in a tile are learned for all states in the tile. Partitioning the state space into many small tiles will slow learning but will improve the probability of finding optimal policies. Conversely, if the tiles are very large then Q-values will be distributed quickly across many states, but there is no guarantee that two states on opposite sides of a tile should share the same action values. In this case, each state may favor a different action, but only one action can be preferred per tile, preventing a correct policy from being found. This tradeoff is mitigated by overlapping layers of tiles to provide both coarse and fine grain generalization. Each observed state updates one tile per layer, and each of these tiles covers a different portion of the state space. The preferred action for a state is the action that maximizes the weighted sum of action values across all tiles that contain that state.

The CMAC algorithm has effectively learned a number of domains including the mountain car and single pole balance [5]. More recently, it has been shown to suffer from some limitations on slightly more complicated problems like the double pole balance [6]. The main difficulty in applying CMAC is choosing a suitable way to break up the state space into tiles. If this is done inexpertly then states that do not prefer the same action can be forced to learn together if they are both confined to a single tile. This will severely slow down, if not prevent, the learning of a successful policy. A secondary concern is the memory requirements for high dimensional scenarios. The number of tiles per mapping scales exponentially in the number of input perceptions for a problem, and storing all visited tiles can quickly become unreasonable. Hashing techniques like those mentioned in [4] can be used to place limits on the memory requirements of CMAC, but they effectively cause non-local generalization of learned values in the event of a hash collision, which can negatively impact policy convergence.

Adaptive Tile Coding Adaptive tile coding (ATC) [3] is a tile coding algorithm that automatically derives variable resolution state abstraction while learning a policy for a specific problem. It is similar to the continuous U-Tree algorithm discussed in [2]. Both methods derive abstractions by starting with a single tile that encompasses the entire state space. Based on observations made while an RL algorithm attempts to learn over the abstract state space, "splits" are introduced. Splits divide individual tiles evenly along feature dimensions into two new abstract states. The idea is to increase the resolution only in areas of the state space where changes in action choices should be made. Splitting continues until the RL algorithm is able to solve the problem using the derived abstract state space. Determining when and where to split tiles is the only significant difference between these methods. Heuristics is used for ATC [3] and a statistical method is used for continuous U-tree [2].

ATC uses two heuristics to determine first when to split and then where to split. The first heuristic keeps track of the lowest Bellman error per time step. If the lowest Bellman update fails to change for a specified consecutive number of updates, *split threshold*, then the heuristic has determined learning has stopped and it is appropriate to split a tile. Once it has been determined that it is appropriate to split, the *policy criterion* heuristic determines where to split. The ATC algorithm updates the Q-values for all potential tiles in the tilings. Every time a potential tile within the current activated tile prescribes a differing action from the activated tile it updates a counter for the potential tile. ATC splits the tile with the potential tile that has the highest counter value to establish that potential tile in the tiling. This process increases the resolution of the tiling in areas where changes in policy are likely.

In [3] it was shown the ATC has a number of advantages over CMAC. First, the tilings are derived automatically, eliminating the need to manually design and discover an effective tiling. Second, ATC was found to be faster at finding π^* than CMAC using the best found parameters for the number of tiles and tilings. The reason for the improvement is that the RL algorithm benefits from the generalization of the overly abstract state space early in the learning. As the abstract state space becomes more specific the new states are already partially learned because they retain the values learned from the more general state they were split from.

Although this approach is an improvement over fixed tile coding methods like CMAC, it suffers from a significant drawback. This approach splits the tiles in half evenly. It is highly unlikely that such a split will be positioned exactly where there is a decision point in which taking one action should be preferred over another. ATC can make up for poor split selections by successively splitting sub-tiles until the decision point is reached. However, many unnecessary states could be introduced and it will slow the RL.

3.1.2 Evolutionary Tile Coding

Evolutionary Tile Coding (EvoTC) is a new approach that takes flexible state space arrangement even further. Like the other adaptive tile coding approaches it starts with a single tile that encompasses the entire state space and introduces splits to increase the detail of the abstraction. The major differences are that EvoTC uses an evolutionary algorithm [7] to determine when and where to place the splits, and the splits can divide tiles unevenly. By dividing tiles unevenly EvoTC should be able to derive more efficient and effective tiling abstractions than other existing tile coding approaches.

In ATC and continuous U-tree the splits are placed in the center of tiles because it is difficult to determine exactly where the optimum split should be made. So, instead they home in on the correct position by adding additional splits. The additional splits are unnecessary and slow learning. EvoTC is able to find better split positions by framing the problem of finding the optimal position and number of splits as an optimization problem where the performance of the RL algorithm is optimized.

EvoTC starts with an initial population of tilings to be evaluated. Each tiling is evaluated independently by pairing it with an RL algorithm that attempts to solve a problem using the tiling as a state abstraction device. The performance of the RL algorithm is considered the fitness of the tiling. Tilings that are more effective at abstracting the state space should enable the RL algorithm to perform better. After all members of the population are evaluated the fittest tilings are kept for successive generations. New tilings based on the fittest members of the previous generation are also introduced into the population for the next generation. The new tilings are generated by applying mutation operators, described later, to the current fittest members of the population. The new population is then evaluated in the same manner the previous generation was. Over the course of generations a tiling should be produced that will enable the RL algorithm to exceed a specified performance threshold and the algorithm terminates.

In the following we provide details on how the tilings are represented in the evolutionary algorithm and how the mutation operators function:

Genetic Representation of Tiles Each chromosome in EvoTC represents a single unique tiling. The chromosomes hold a tile arrangement described as a binary decision tree. The genes that make up the chromosome describe the nodes in the tree. Leaf nodes represent a current tile and hold the Q-values associated with the abstract state. Non-leaf nodes represent tile divisions and describe along what feature the division is made and its position. See figure 1 for an illustration of how the tiling is represented as a tree. The genetic representation is non-fixed to enable the tiling to become more complex as needed. The process of how the chromosome is extended is described in the *divide* mutation operator description.

Mutation Operators The key to the EvoTC is its mutation operators which make diverse tile arrangements in the search for the optimal arrangement. These mutation operators are applied, with a specified probability, to existing chromosomes in the population to make new chromosomes at the end of each generation. Two mutation operators are used in this algorithm:

- The *shift* operator moves the position of tile splits. The purpose of this mutation operator is to explore the ability of the existing tiling arrangement to properly abstract the state space. As such this operator should be activated with a higher probability than the *divide* operator which changes the structure of the tiling arrangement.

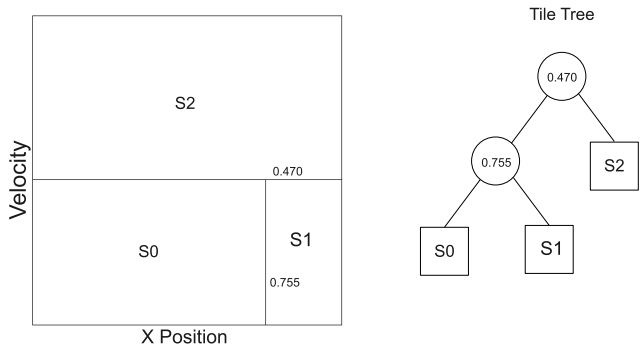


Figure 1: This figure illustrates how EvoTC represents the tile discretizations of a state space as a binary decision tree. Left: shows a sample discretization of the two dimensional state space of the mountain car problem. Right: shows the corresponding decision tree which is used to find the Q-values associated with the individual tiles.

When this mutation operator is activated it selects a number of division nodes to be modified at random. For each selected node, the position of the divide is shifted by a small amount determined by a Gaussian random distribution up to within 1% of the edge of the tile. This prevents a pair of adjacent tiles from effectively becoming one tile if one of the tiles holds 0% of the state space. After the selected genes are altered, the tree is updated with the mutated genes.

- The *divide* operator introduces new splits to the tiling to add granularity to the abstract state space. It should have a relatively low probability of being activated to give the *shift* operator sufficient time to explore more general tilings.

  This operator functions by selecting a single leaf node at random to divide. The node is divided by randomly selecting a dimension to divide along and the division is placed using a random Gaussian distribution over the center of the tile. Once the divide is set, new leaf nodes and genes are created and attached to the new divide node. Finally, the Q-values for the new leaf nodes are initialized to a value that encourages exploration of the new tiles.

3.2 Continuous State Space Abstraction

Given a Markov Decision Problem (MDP) defined over a set of states $\mathcal{S}$ and actions $\mathcal{A}$, reinforcement learning (RL) algorithms seek to learn an optimal policy π^* which selects the appropriate action $a \in \mathcal{A}$ for each state $s \in \mathcal{S}$ to reach some specified goal state. Typical reinforcement algorithms learn π^* by repeatedly experiencing states leading to the goal state a number of times. In domains with a continuous set of states, the probability of repeatedly visiting *any* state approaches zero, preventing the learner from converging to π^* . State space aggregation or abstraction techniques must then be introduced to allow learning of an optimal policy by turning the continuous space $\mathcal{S}$ into a discrete space $\mathcal{S}'$.

State space abstraction techniques can be classified into five categories depending on the "coarseness" of the abstraction and what components of $\pi*$ in the original state space are to be preserved in the abstraction [8]. In that work, Li et al. proved that the optimal policy learned on several of the categories of abstractions (model-irrelevance, Q^π-irrelevance, and Q^*-irrelevance) resulted in an optimal policy in the original space. However, the two abstraction categories that produce

the sharpest reduction in the size of the state space are not guaranteed to learn a policy that will converge to the optimal solution. This makes applying these powerful abstraction techniques (a^*-irrelevance and π^*-irrelevance) dangerous in general, as they might prevent the learner from arriving at the optimal policy.

One of these classes of abstractions, the a^*-irrelevance abstraction, which groups two base states together if they share the same optimal action, is of particular interest. One of the most popular types of abstraction techniques, tiling, falls into this category. Examples of common tile based abstractions include CMAC [4], and U-tile distinction [9, 2]. While tile based methods have been shown effective in a number of situations, there are serious drawbacks to using them effectively. Engineering a tiling is typically done by hand, and it can be very difficult to find an appropriate tiling for a given problem or to correctly set the parameters in methods that build a tiling during the learning process. It has also been shown that tiling techniques cannot solve some standard benchmark problems [6].

Here we propose and evaluate three *automatic* tiling methods that efficiently learn how to abstract a space but still allow a learner to converge to the optimal policy. These abstractions are applied to the one dimensional state space produced by RL-SANE algorithm [10] which allows us to focus on the methods without dealing with the dimensionality of the original state space. Each of these methods allows the abstract states that are used by the learner to be redrawn in an effort to get the states that share the same optimal action to fall into the same tile, and improve the speed of the learning. It should be noted that the neural network that is involved in the process is also evolving over time and is able to learn optimal policies on difficult problems even when using a fixed tiling, however, we seek to reduce the burden on the neural network in the learning process by using a better tiling approach. It is thought that once good automatically-tiling routines are identified in this learning process, they can be used to abstract higher dimensional state spaces, cutting out the need for a dimensionality-reduction technique.

We show via empirical study on two well-known RL benchmark problems that each of the three automatic tiling methods proposed here allow the learning algorithm to significantly improve its rate of convergence when compared to the base RL-SANE algorithm using a fixed tiling. Additionally, we show that the automatic methods we propose here result in very few abstract states (tiles) being used in order to learn the test problems.

The rest of this section is organized as follows: First we describe related tile encoding methods and give background on the RL-SANE algorithm. The three abstraction techniques are described in depth in the Automatic Methods section. The methods are then applied to the two problems given in the Experimental Setup section, and the results of this study are given in the Experimental Results section. Finally, the section concludes with a summary of our contributions and possible future directions.

3.2.1 Background and Related Work

Here we give some necessary background on various tile encoding methods that have appeared in the literature, as our methods can be interpreted as tile encodings as well. We also provide details on the RL-SANE algorithm, the platform we use to reduce the dimensionality of the given learning problems and on which we examine our proposed methods.

Tile Encoding Many tile encoding methods exist in the literature, and several popular mechanisms are variations on the Cerebellar Model Articulation Controller (CMAC) algorithm [4] which generalizes the learning of one state to a set of nearby states and is based on how the human brain is thought to respond to stimuli [1].

In tile encoding methods, the state space can be thought of as being broken apart into a number of tiles, and every time one state is observed, all the other states that belong to the same tile (or tiles, in the common event of overlapping layers of tiles) also experience the learning rewards. The size of the tiles controls the resolution of the abstraction, smaller tiles result in a finer resolution, but cause more states to be learned before the RL can find π^*. The location in the state space where the boundaries occur in the tiling can have a great impact on the ability of an algorithm to learn π^*. For example, if in order to solve a problem, action a_1 must be taken from state s_1 and a_2 must be taken from s_2, then if s_1 and s_2 fall on the same tile (same abstract state). The optimal policy will be impossible to learn since only one action can be taken from the abstract state.

A fundamentally different approach to the tile encoding problem is taken in [9] with the U Tree algorithm and later extensions to this work [2] to have it work in continuous domains. Here, the coarseness of the abstraction is not fixed *a priori* by setting a tiling, but rather, the space is viewed as a single tile and then repeatedly split in areas where it is determined a finer abstraction is needed. The algorithm decides to spilt a tile in two when each of the subtiles shows a different distribution of observations than the whole tile, indicating that more information about the problem can be gained by splitting the tile. These approaches allow for automatic construction of a state space abstraction; however, they still suffer from a fundamentally arbitrary splitting mechanism.

When the U Tree algorithm finds that more resolution can differentiate observations in one area of the state space, the algorithm simply splits the tile in two at the center of the tile. Splitting a tile into two halves will lead to problems when the section of the state space that needs more resolution is towards the center of a tile, as many splits will need to take place before this area achieves the necessary resolution. This causes the abstraction to include and subsequently to learn many unnecessary tiles which hinders the power of the reduction. In contrast, the methods we propose allow the split locations to be positioned according to the needs of the problem and not in a pre-specified location, nor deduced by a human prior to applying a learning algorithm.

RL-SANE The RL-SANE algorithm is a powerful reinforcement learning and state abstraction algorithm [10]. It combines a neuroevolution approach to constructing neural networks [11] with a fixed tiling over a one dimensional abstract state space to allow a learner to efficiently learn complex problems by learning the optimal action for each tile in this abstract space. An overview of this process is given in Figure 2.

For any dimensionality of input space $\mathcal{S}$ the artificial neural network layer of RL-SANE takes the input measured across m dimensions and reduces it to a single output value $z \in [0, 1]$ corresponding to a single abstract state $s' \in \mathcal{S}'$. This one dimensional output space can still represent infinitely many states, so a tiling is applied to it. The fixed tiling simply splits $\mathcal{S}'$ into a number of equal sized tiles with no consideration given to the exact position of where each split occurs, meaning that the same problem that occurs with the fixed tiling methods can occur here as well.

The difference between a true tile encoding technique and the RL-SANE algorithm is the fact that if a given abstraction does not allow π^* to be found, the Artificial Neural Network (ANN) can mutate via neuroevolution and cause a different distribution of outputs which may better suit the tiling and allow learning to continue. The original RL-SANE algorithm included a user specified parameter β to determine the number of tiles to lay over $\mathcal{S}'$. Evidence displayed in [10] shows that the algorithm's overall convergence is not very sensitive to β and the rate of learning can be significantly impacted by a poor selection.

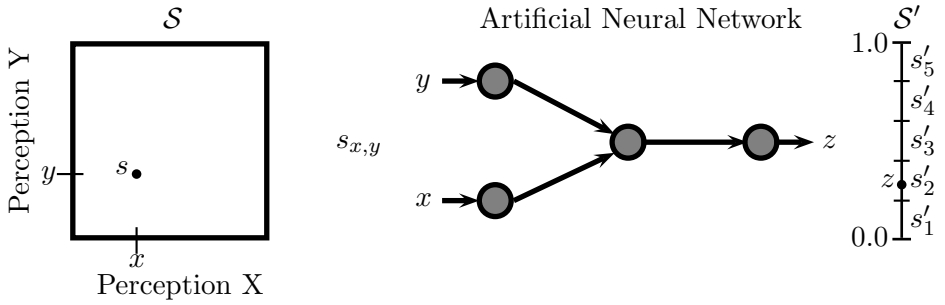


Figure 2: Overview of the RL-SANE algorithm transforming a ground state $s_{x,y}$ in a sample two dimensional state space $\mathcal{S}$ to the abstract state s'_2 in the one dimensional abstract state space $\mathcal{S}'$.

3.2.2 Automatic Abstraction Methods

In this work we focus on three different types of automatic methods that are able to redraw the abstraction boundaries of $\mathcal{S}'$ online as the learner embedded in the RL-SANE algorithm is learning. Here we describe mutation, Maximum Density Separation, and Temporal Relative Extrema methods for automatic state abstraction.

Mutation The mutation method of automatic state bound construction makes direct use of the neuroevolution process that is at the heart of the RL-SANE algorithm, and is the closest method to fixed tiling of the three methods we discuss in this section. As mentioned above, the basic RL-SANE algorithm takes the number of abstract states to generalize to, β, as a parameter. The mutation method encodes β as another gene in the chromosome and allows it to be mutated during the evolution of each neural network. This allows the evolutionary process to automatically explore different state abstraction possibilities in an effort to find a new one that better groups similar states together based on the output of the neural network.

In this work, we experimented with two variations of this idea. The first allows the number of abstract states to increase or decrease by one per mutation, and the second allows the number of states to change by a random amount up to a user defined threshold in a single mutation. Whenever a mutation to the state bound occurs, the method redistributes the previously learned Q-values for each action in each abstract state into the new abstract states in proportion to the overlap between the old and new states. For example, if there are half as many new states as old ones, then each new state gets initial Q-values that are the averages from the two old states that the new one overlaps. This enables the learner to reuse values that it had previously learned while allowing more refined abstractions to come into existence and drive the learner to a better solution.

This method directly improves the situation of estimating the proper fixed number of states, achieving our goal of automatic state abstraction; however, there are still some drawbacks that need to be addressed. While the boundaries are redrawn according to the chance of a mutation to the neural network, they are still arbitrarily placed over the space; that is, each state in the abstraction cover the same portion of the space, and there is no intuition that implies that this is a good strategy in general. We would rather have the smaller states be introduced where finer resolution is needed, and broader states where a coarse abstraction would do.

Maximum Density Separation The Maximum Density Separation (MDS) approach takes a different tack in determining the tiling to be used. Unlike the mutation method, MDS can place the boundaries of a split anywhere in the state space and can add or remove as many abstract

Algorithm 1 MDS (Maximum Density Separation)

required: number of bins for frequency distribution
output: new abstract state mapping
≪embedded within a reinforcement learning algorithm≫
get next state s' by following π^* from state s
if s' != fail state
  $s := s'$
  increment frequency distribution (s)
else if s' == fail state
  locate relative extrema in frequency distribution
  erase old partitions of $\mathcal{S}'$
  partition $\mathcal{S}'$ in the center of two relative maximums
    if a relative minimum occurred between them
end else if

states at a time as the algorithm determines necessary. Similar to that method, the previously learned Q-values in each of the old tiles are blended together to form the initial Q-values of the new tiles. This method intuitively views dense clusters of observations as belonging to a single state, and abstracts the state space so that these dense clusters are located on separate tiles from one another. The split between tiles occurs at the farthest point between two dense regions of observations. This approach is principled by the idea that nearby states will prefer the same action, however the size of each these groups may vary so we must use an adaptable partitioning solution.

An overview of the Maximum Density Separation method is given in Algorithm 1. For a single run of the problem in a given RL algorithm, this method records the frequency of observations across the state space until a failure or the goal state is reached. On a failure, the constructed frequency distribution is searched for relative extrema, using a soft-thresholding approach to prevent small fluctuations in the distribution from leading to many spurious extrema. Once the relative extrema have been identified, a partition is placed in the space in the center of every two relative maximums, as long as a relative minimum occurred between them. The splits between abstract states are made in this fashion in accordance with the maximum margin principle [12], which seeks to minimize the structural variance in a hypothesis. Positioning the splits as far as possible from the dense regions of observations minimizes the risk that in the next run of the problem new observations belonging to one dense region will spill into an adjacent state and mislead the learning there. We wish to make splits as far from the dense regions of points as possible since the next set of observations may move the peak slightly from where it was found in the previous generation, but the same observations will still prefer the same action and should not fall into adjacent tiles as would be the case if the partitions were placed nearer to the dense centers. This process is linear in the number of bins used to measure the frequency distribution, and in practice had only a negligible impact on the running time of each generation of the algorithm, and so is a feasible abstraction algorithm in terms of time complexity. After the new state abstraction has been set, the Q-values that were learned on the earlier state abstraction are transferred to the new abstraction in the same manner as described in the mutation method.

This method effectively overcomes two of the perceived limitations of the mutation method: abstract state partitions can be placed anywhere in the space and the number of states in the space does not directly depend on the previous number. The MDS method does introduce some other limitations, however. It could be the case that an area of dense observations are not really homogeneous in terms of preferred action, but are coincidentally grouped together by the ANN. In this case, the abstract states might still become successful if the ANN adapts and separates

these states into two different clusters in a later evolutionary stage. Another limitation of MDS is that it has no notion of observations that led to the failure, and the series may easily be grouped with other observations with similar values but should be separated and allowed to pursue other actions as soon as possible. The MDS method makes no provision for this possibility and relies on the ANN to separate out the other states in a later generation. The next method addresses these shortcomings.

Temporal Relative Extrema As the name implies, the Temporal Relative Extrema (TRE) method, summarized in Algorithm 2, creates an abstraction by incorporating the order in which observations are generated and not just their values as in the MDS method. Like the MDS, TRE is capable of partitioning $\mathcal{S}'$ into as many abstract states as necessary, and can place partitions between states anywhere in the space. During preliminary studies, we noticed that the observations in $\mathcal{S}'$ frequently followed a periodic function, much like a sine curve, if states were viewed in the order they were observed. The TRE approach was created to encourage more exploration in the learner and break away from the periodic repetition of known states to get the learner to visit new and possibly more beneficial states.

Per the TRE method, all of the observations are recorded for a single run of the problem until a failure is encountered. At this point, the algorithm iterates through each stored observation and identifies relative maxima and minima as those places are associated with restarting the next period of observations. Each relative extrema are compared to the rest and if they are within a user defined threshold t away from each other then they are considered to be the same extrema and are merged together. For minima, the greatest (innermost) observation is stored after the merge, and for maxima, the smallest (again innermost) observation is stored. The state space $\mathcal{S}'$ is then partitioned at each of the final extrema locations and becomes the new abstraction for the next generation. The initial Q-values of the new abstraction are taken from the previous abstraction in the same manner as the mutation and MDS methods. The time complexity of the TRE method is linear in the number of observations for each run of the problem, and since the observations must be generated anyway, there is no noticeable effect on the overall running time of the learning algorithm.

The heuristic of partitioning the state space based on where the learner begins to repeat observations for the same problem is quite distinct from the previous approaches mentioned here. In effect, this groups the heavily repeated observations into the same abstract state while allowing for the relative extrema to more easily find their own preferred actions, which can lead to an improved learning rate. If the extrema prefer the same action as the other heavily repeated observations then there is not much harm done by separating them, as their initial Q-values will be shared according to the previous abstraction anyway and should not hurt the overall learning rate. The similarity parameter t does not need to be significantly tuned; it suffices to set it small compared to the range of possible values for an observation. If t is very small (s.t. $|z_i - z_j| > t$ for nearly all observations $z_i, z_j \in \mathcal{S}'$ near relative extrema, with $i \neq j$), many abstract states will be created near the extrema, but this has little real impact on the learning since they will generally share the same Q-values over successive generations.

3.3 Density-based Automatic State Space Abstraction

There are many ways to achieve state space abstractions. In[8], the authors constructed a five tier categorization scheme for them depending on the "coarseness" of the abstraction and what parts of the optimal policy π^* get carried over from the original space to the abstract one. For three levels of coarseness (model-irrelevance, Q^π-irrelevance, and Q^*-irrelevance) they prove that an optimal

Algorithm 2 TRE (Temporal Relative Extrema)

required: similarity threshold t for comparing extrema
output: new abstract state mapping
≪embedded within a reinforcement learning algorithm≫
get next state s' by following π^* from state s
if s' != fail state
  $s := s'$
  record s
else if s' == fail state
  for each relative extrema
    merge extrema if closer than t from an existing extrema
    otherwise store the extrema
  end for each
  erase old partitions of $\mathcal{S}'$
  partition $\mathcal{S}'$ on the inside each stored extrema
end else if

policy found in the abstract space $\mathcal{S}'$ will also result in an optimal policy in the ground state space $\mathcal{S}$. The other two categories (a^*-irrelevance an π^*-irrelevance) do not share this same guarantee, though are in some ways more valuable because they can abstract down the size of the state space significantly better than the other three methods.

One of these classes of abstractions, the a^*-irrelevance abstraction, which aggregates two base states together if they share the same optimal action, is the focus of this paper. One of the most popular types of abstraction techniques, tiling, falls into this category. Examples of common tile based abstractions include CMAC [4], and U-tile distinction [9, 2]. These methods aggregate ranges of a continuous state space to discrete abstract states, resulting in a finite environment for a RL algorithm to learn in. While tile based methods have been shown effective in a number of situations, there are serious drawbacks to using them. Engineering a tiling is typically done by hand, and it can be very difficult to find an appropriate tiling for a given problem or to correctly set the parameters in methods that build a tiling during the learning process. Automatic tiling methods [3], that construct tilings during the learning process, have shown to be very sensitive to parameter choices. Recently, tiling methods have been shown ineffective in the double pole balance setting, a challenging RL standard problem [6].

Here we propose and evaluate an *automatic* tiling method called maximum density separation that efficiently learns how to abstract a space but still allows a learner to converge to the optimal policy. This abstraction is applied to the one dimensional state space produced by the RL-SANE algorithm [10] which allows us to focus on the aggregation technique and not feature space dimensionality reduction. This method utilizes the distribution of observations to choose more intuitive abstract states to improve the speed of learning. It should be noted that the neural network that is involved in the process is also evolving over time and is able to learn optimal policies on difficult problems even when using a fixed tiling, however, we seek to reduce the burden on the neural network in the learning process by using a better tiling approach. Effective automated tiling techniques will eliminate the difficult manual process of designing state abstractions. When used in conjunction with feature space dimensionality reduction techniques, automated state aggregation methods will greatly increase the scalability and applicability of existing RL algorithms.

We show via empirical study on the mountain car and double pole balance benchmark RL problems (Figure 3) that the maximum density separation algorithm proposed here allows the learning algorithm to significantly improve its rate of convergence to π^* when compared to the

base RL-SANE algorithm using a fixed tiling and another more simplistic automated approach. Additionally, we show that our automatic method is capable of creating a more compact abstract space than a traditional fixed tiling approach.

In the next section we give background on the RL-SANE algorithm we use for dimensionality reduction. Next, the maximum density separation algorithm is introduced and explained in detail. This method is then applied to the two problems given in the Experimental Setup section, and the results of this study are given in the Experimental Results section. Finally, the section concludes with a summary of our contributions and some future directions.

3.3.1 Background

For any dimensionality of input space $\mathcal{S}$ the ANN layer of RL-SANE takes the input measured across m dimensions and reduces it to a single output value $z \in [0, 1]$ corresponding to a single abstract state $s' \in \mathcal{S}'$. This one dimensional output space can still represent infinitely many states, so a tiling is applied to it. The fixed tiling simply splits $\mathcal{S}'$ into a number of equal sized tiles with no consideration given to the exact position of where each split occurs. The tile boundaries are suboptimal since no consideration is given to how observations will gather in the space. However, the ANNs used in this approach are not fixed. They are produced and adapted through the use of the NEAT neuroevolutionary algorithm [11]. The NEAT algorithm adapts the ANNs to fit the observed ground states to the structure of the abstract state space making the placement of tile boundaries less critical. This ability of RL-SANE to adapt to the structure of a specified abstract state space improves its ability to discover π^* over other tile coding approaches. The original RL-SANE algorithm included a user specified parameter β to determine the number of tiles to lay over $\mathcal{S}'$. Evidence displayed in [10] shows that the algorithm's overall convergence is sensitive to β. To overcome the limitations of a fixed tiling we propose here an alternative which can work to provide better tiling layouts across the space and allow the RL-SANE algorithm to quickly converge to an optimal policy without the need for careful estimation of an ideal β parameter.

3.3.2 Maximum Density Separation

The MDS method can place the boundaries of a split anywhere in the state space and can add or remove as many abstract states at a time as the algorithm determines necessary. Similar to that method, the previously learned Q-values in each of the old tiles are blended together to form the initial Q-values of the new tiles. This method intuitively views dense clusters of observations as belonging to a single state, and abstracts the state space so that these dense clusters are located on separate tiles from one another. The split between tiles occurs at the farthest point between two dense regions of observations. This approach is principled by the idea that nearby states will prefer the same action, however the size of each these groups may vary, so we must use an adaptable partitioning solution.

There are two main challenges related to the MDS approach. The first is how to estimate the density of the space or otherwise cluster observations, and the second is deciding the appropriate time to reassess the state space. In the context of the first problem, MDS can be thought of as a framework where an appropriate clustering or density estimation technique can be applied at the discretion of the user. For example, it might be known that the states occur in very tight clusters, so a clustering algorithm that is designed to find such clusters should be used in this case. There are numerous clustering approaches that can be incorporated; the reader is directed to a recent survey on the field for more information [13]. If specifics about the distribution of observations are unknown, then kernel density estimation can likewise be used to find the peaks of the sample

Algorithm 3 Maximum Density Separation (MDS)

required: number of bins for frequency distribution
output: new abstract state mapping
≪embedded within an RL algorithm≫
get next state s' by following π^* from state s
if s' != fail state
  $s := s'$
  increment frequency distribution (s)
else if s' == fail state
  locate relative extrema in frequency distribution
  erase old tiles of $\mathcal{S}'$
  partition $\mathcal{S}'$ in the center of two rel. maximums
end else if

(a) (b)

Figure 3: Depictions of the mountain car (a) and double pole balance (b) problem domains.

density distribution. One candidate procedure for this purpose is the mean shift algorithm [14], but there exist many other options. In this work, we model the density of the abstract state space using a simple histogram approach as it does not require much additional computation to employ within the learning algorithm. While more complicated clustering or density estimation methods can be employed, the tradeoff in terms of possibly more accurate state boundary identification must be balanced against the time cost of performing the analysis.

Determining when to repartition the abstract space can have a large impact on the convergence speed of the learning algorithm. If the tiles are repartitioned too frequently the learner may not have time to learn accurate Q-values on the abstract states, causing the algorithm to converge to a less than optimal policy. On the other hand, attempting to learn in a poorly partitioned space can lead to wasted update cycles as the Q-values do not give meaningful direction to the learner. In our implementation of MDS we repartition the space whenever a failure state is reached by the learner since it is possible that a new partitioning might help the learner avoid failing the problem in a subsequent attempt.

An overview of the MDS method is given in Algorithm 3. For a single run of the problem in a given RL algorithm, this method records the frequency of observations across the state space until a failure or the goal state is reached. On a failure, the constructed frequency distribution is searched for relative extrema, using a soft-thresholding approach to prevent small fluctuations in the distribution from leading to many spurious extrema. Once the relative extrema have been identified, a partition is placed in the space in the center of every two relative maximums. The splits between abstract states are made in this fashion in accordance with the maximum margin principle [12], which seeks to minimize the structural variance in a hypothesis. Positioning the

splits as far as possible from the dense regions of observations minimizes the risk that in the next run of the problem new observations belonging to one dense region will spill into an adjacent state and mislead the learning there. This process is linear in the number of bins used to measure the frequency distribution, and in practice had only a negligible impact on the running time of each generation of the algorithm, and so is a feasible abstraction algorithm in terms of time complexity.

3.4 Incremental Feature Selection

This work is about automated feature selection for RL. Although feature selection has been extensively studied for supervised learning [15, 16], existing methods are either inapplicable or impractical in the RL setting. Filter methods rely on training data, which is not available in RL, to select features. Wrapper methods require repeatedly executing a learning algorithm on each candidate feature subset, and are impractical for RL due to their prohibitively high computational and sample cost. A promising approach is to embed feature selection into the training process of a learning algorithm. However, the embedded approach has to be tailored for the learning algorithm of interest.

In this section we describe an embedded incremental feature selection algorithm for a neuroevolutionary function approximation algorithm NEAT (NeuroEvolution of Augmenting Topologies) [11], which we call IFSE-NEAT. The main idea of IFSE-NEAT is to embed incremental subset selection into the neuroevolutionary process of NEAT. Instead of evolving networks with the full set of features as NEAT does, IFSE-NEAT initializes networks with one feature. IFSE-NEAT then iteratively adds features to the current best network that contributes most to its performance improvement while evolving the weights and topology of that network.

3.4.1 Background

Prior to this work, feature selection for reinforcement learning has focused on linear value function approximation [17, 18] and model-based RL algorithms [19]. For neuroevolution algorithms such as NEAT, only random search has been explored [20]. In this light we can see that IFSE-NEAT is a novel approach in feature selection for RL.

Our experimental study has shown several promising results for IFSE-NEAT. We find that the algorithm is nearly unaffected in its ability to select relevant features as the number of irrelevant features grows very large. This, in turn, allows for a better policy to be derived than NEAT. Additionally, by using only a few relevant features we are able to learn a good policy while limiting model complexity.

3.4.2 Combining Feature Selection and Genetic Policy Search

NEAT Neural networks (NNs) are efficient function approximators that can model complex functions to an arbitrary accuracy. The drawbacks of using NNs in RL domains have been that NN design was a difficult manual process and training was a supervised learning process. Neuroevolutionary approaches, which utilize genetic algorithms to automate the process of training and/or designing NNs, eliminate these drawbacks, allowing NNs to be easily applied to RL domains. NeuroEvolution of Augmenting Topologies (NEAT) is a novel RL framework based on neuroevolution. By evolving both the network topology and weights of the connections between network nodes, NEAT solved typical RL benchmark problems several times faster than competing RL algorithms with significantly less system resources [11].

However, one limiting issue with NEAT is that it assumes that all features provided by the environment are relevant and necessary, and attempts to incorporate all the features into its solution

Algorithm 4 IFSE-NEAT(N,k,L,p)

```
1: //N: set of all available features
2: //k: number of features to select
3: //L: number of generations to evolve
4: //p: population size
5: BACKBONE ← outputNodes //initialize the BACKBONE
6: SELECTED_SET ← null //initialize the selected feature set
7: for i ← 1 : k do
8:     BEST_NETWORK ← null
9:     BEST_FEATURE ← null
10:    //iterate through all candidate features outside SELECTED_SET
11:    for q ← 1 : N − i do
12:        //create new network N_b based on candidate feature F_q
13:        N_b ← COMBINE( F_q, BACKBONE )
14:        //create a population of p networks based upon N_b
15:        population ← INITIALIZE-POPULATION(N_b, p)
16:        //evolve population using NEAT for L generations
17:        for j ← 1 : L do
18:            NEAT-EVOLVE(population)
19:        end for
20:        //select the champion from population
21:        champion ← BEST-QUALITY(population)
22:        if champion > BEST_NETWORK then
23:            BEST_FEATURE ← F_q
24:            BEST_NETWORK ← champion
25:        end if
26:    end for
27:    Add BEST_FEATURE to SELECTED_SET
28:    BACKBONE ← BEST_NETWORK
29: end for
```

networks. The extraneous features will unnecessarily complicate the networks and severely slow the rate at which NEAT is able to derive an effective policy. In the following section we describe a new algorithm based upon NEAT that builds a small set of required features while learning an effective policy.

Incremental Feature Selection Embedded in NEAT (IFSE-NEAT) To deal with the exponential search space, we adopt sequential forward search (SFS), an efficient search strategy which has proven effective in finding near-optimal subsets in supervised feature selection. Starting from an empty set, SFS iteratively adds one feature at a time to the current best set until a desired number of features k are selected. Since in each of the k iterations, it goes through all N features outside of the current best set, the time complexity of SFS is $O(kN)$. Although SFS does not guarantee the optimal solution, it is capable of selecting relevant features while keeping irrelevant or redundant features out of the final subset. The method is particularly suitable for high-dimensional problems where large portions of the features are irrelevant or redundant.

Algorithm 4 provides a basic overview about how IFSE-NEAT functions and is able to select a minimal set of features. IFSE-NEAT incrementally adds features to a NN that we call the $BACKBONE$. The $BACKBONE$ network utilizes the best discovered feature set and represents the current best derived policy. It is persistent through additions of new features to the feature set and it is what makes IFSE-NEAT an embedded algorithm as opposed to a straightforward wrapper algorithm.

Initially, the $BACKBONE$ network consists of only the output nodes (line 5). Then, for each of the individual features available, F_q, a NN is generated by connecting a single input node to every output node (line 13). In parallel, or independently, a population of networks based upon a single-

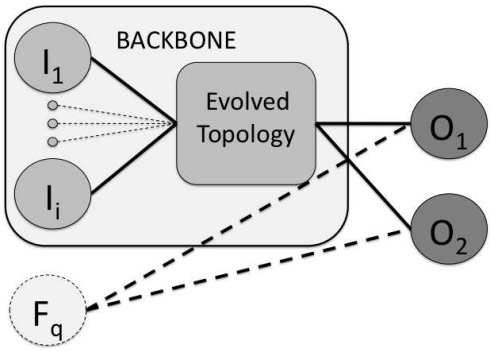


Figure 4: This figure illustrates how a candidate feature is incorporated into the current $BACKBONE$ network to create a new base candidate network, N_b. The new feature F_q is introduced to the network and is provided connections, the dashed lines, to every output node. $BACKBONE$ in this figure represents the best evolved solution network using only the previously selected features.

input base of networks is generated. Each network in the population share the topology of the base network, but has randomly generated weights on the edges joining the nodes. The population of NNs are then evolved via the standard NEAT algorithm for L generations (lines 17-19). At the end of the NEAT process, the champion of each population (the network representing the best policy) is identified. The champions (each corresponding to a candidate feature F_q) are then compared against one another to decide the $BEST\_NETWORK$ and $BEST\_FEATURE$ (lines 22-25). It is our hypothesis that the best performing network, $BEST\_NETWORK$, will point to the most relevant feature. Therefore, the $BEST\_FEATURE$ that produced the $BEST\_NETWORK$ is then added to the $SELECTED\_SET$ (line 27), and the $BEST\_NETWORK$ becomes the $BACKBONE$ (line 28) for subsequent iterations where the algorithm will determine the next features to add to the feature set.

In the subsequent iterations the remaining features are independently combined with the $BACKBONE$ network and then re-evaluated. As in the first feature selection iteration, new populations of NNs, random variations of the base networks, are again evolved by NEAT for L generations. The algorithm stops once a desired number of features are selected. Alternatively, the algorithm can stop when one of the populations produces a network that represents a suitable solution to the problem.

The process for combining the $BACKBONE$ network, COMBINE(F_q,$BACKBONE$) (line 13), is illustrated in Figure 4. In this process a new base network N_b is created for a candidate feature F_q by connecting F_q to each of the output nodes. The weights of these new edges are assigned zero to preserve the policy of the $BACKBONE$ network. Preserving the policy of the $BACKBONE$ network bootstraps the successive networks and improves IFSE-NEAT's ability to determine the relevance of potential new features.

Analysis of Algorithm 4 shows that time complexity of IFSE-NEAT is $O(kN)$ times the NEAT process, which itself is dependent on the population size p and the number of generations L. In practice, however, we can do better than this. For the first few selected features, L can be very small and the algorithm can still identify relevant features, allowing a significant speedup to roughly $O(N)$ times NEAT. Once the $BACKBONE$ network is reasonably fit, L must be increased to allow new features enough time to have an impact in the more complex network.

3.5 Sample Aware Feature Selection

Here we propose Sample Aware Feature Selection-NEAT (SAFS-NEAT), an embedded feature selection algorithm for RL. Intuitively, this method seeks to restrict the learner to a low-dimensional view of the environment, and incrementally expands this view to include new dimensions as observed samples expose potentially relevant features. Since learning algorithms generate samples from the environment during their execution, a large data set of samples can be gathered quickly and easily. By further exploiting the structure of the samples, a class label can be designated and supervised feature selection algorithms can then be brought to bear on the problem. As indicated above, this procedure is embedded in the NeuroEvolution of Augmenting Topologies (NEAT) genetic policy search algorithm [11]. NEAT was chosen for its ability to evolve policies in continuous state and action spaces without the need for additional abstraction devices. Additionally, forward selection of relevant features fits well with the complexification concept of the NEAT algorithm: start with a minimal complexity solution and evolve additional components as necessary.

We demonstrate the effectiveness of SAFS-NEAT in two challenging simulation environments. The first is a racing domain described by continuous state and action spaces. The racing domain requires the agent to make precise control decisions while knowing only what is visible from the driver's perspective. Second, a more challenging variant of the classical RL pole balancing domain, the double inverted pendulum balancing problem [21], where the agent must learn a complex policy in a continuous state space using discrete actions to control the motions of a cart in order to balance two poles. We empirically show that SAFS-NEAT is able to learn near optimal control policies in both environments even as the dimensionality of the environments increases. The algorithm also outperforms a competing feature selection algorithm, FS-NEAT [20], in these domains, while not requiring more samples than the basic NEAT algorithm.

The rest of this section is divided between relevant background material to the approach in Section 3.5.1 and the embedded feature selection algorithm itself in Section 3.5.2.

3.5.1 Background

Reinforcement learning (RL) is designed to train agents to make optimal choices in sequential decision making problems. The problem environment can be compactly expressed as a Markov Decision Process (MDP) described the tuple $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$. In this work we consider problems that describe factored state spaces $\mathcal{S}$ such that $\mathcal{S} = \{S_0 \times S_1 \times \ldots \times S_n\}$ for a problem with n state variables. Similarly, we consider the action space $\mathcal{A}$ to be factored as well. For every point in the state space $\mathbf{s} \in \mathcal{S}$ a point in the action space $\mathbf{a} \in \mathcal{A}$ must be selected by the agent. Each state $\mathbf{s}$ is an n dimensional vector $\langle s_0, s_1, \ldots, s_n \rangle$ where each component s_i takes a value in the domain of its respective state variable (feature) S_i. The notation and semantics for each action $\mathbf{a}$ is analogous to this description. The transition function $T(\mathbf{s}'|\mathbf{s}, \mathbf{a})$ governs the dynamics of the environment, or the probability of arriving in state $\mathbf{s}'$ after taking action $\mathbf{a}$ in state $\mathbf{s}$. The reward function $R(\mathbf{s}, \mathbf{a})$ gives the agent a reward value $r \in \mathrm{R}$ to every transition. Both T and R are assumed to be unknown by the agent in our work. As the agent explores the environment, the reward values seen allow the agent to build a value function which indicates the expected value of taking a particular action in a particular state based on past experience [22, 23]. The agent seeks to learn a policy function, $\pi : \mathbf{S} \mapsto \mathbf{A}$ which maximizes the aggregate sum of rewards from any start state to the goal. Such a function is called the optimal policy π^*, and is the learning target for agents in our problem formulation.

Feature selection seeks to reduce the number of variables in a problem while still permitting an optimal (or near optimal) learning model to be constructed. Existing comprehensive surveys

categorize feature selection algorithms for supervised learning into filter, wrapper, and embedded approaches based on their subset evaluation method [16, 15]. Filter based methods use intrinsic data characteristics to guide the selection process, and with few exceptions [24, 25], have been largely unstudied in the RL domain due to the lack of an initial data set and a clear replacement for the supervised class label. The methods described in [24, 25] place restrictions on the samples used which make them unsuited for on-line feature selection. Wrapper based methods construct learning models from competing candidate feature subsets and the performance of the learned models are used to rank the subsets. These methods are prohibitively expensive to apply in RL due to the number of models (policy functions) that must be constructed. Embedded approaches perform feature selection during the model building phase, leading to a single learned model using a subset of features. This approach is readily adaptable to the RL domain and is the structure that we adopt here.

While any RL algorithm could potentially be the target of an embedded feature selection algorithm, we elected to use the direct policy search algorithm NeuroEvolution of Augmenting Topologies (NEAT) for its ease of use and ability to handle both discrete and continuous state and action spaces without additional abstraction layers [11]. Here we give a brief overview of the NEAT algorithm, more details can be found in [26]. NEAT uses neural networks (NNs) to efficiently approximate policy functions, where the input layer is provided the current state and the output layer produces an action to be taken in that state. NEAT begins with a population of simple perceptron networks and gradually builds more complex ones through a process called *complexification*. In every generation of the evolutionary process, the networks in the population are evaluated based on the quality of the policy they produce, and those demonstrating the best quality (or fitness) survive into the next generation. Derivative networks based upon the surviving networks are generated by the mutation operators. These mutation operators modify the weights of the edges and even add topological elements such as new nodes and connections. These operators are activated with varying probability over the population of networks. The results of NEAT are NNs that are automatically generated, not overly complicated in terms of structure, and custom tuned for the problem at hand. A number of parameters control the evolutionary process and are discussed in Section 4.5.1.

Feature Selective-NEAT [20] (FS-NEAT), an early approach for on-line feature selection, differs from NEAT in that all NNs in the initial population start with a single connection between a randomly selected input and output. Features are randomly attached to the NNs in subsequent generations, allowing the algorithm to stochastically arrive at a potentially good set of features during evolution. The evolutionary process treats the inclusion of another input node the same as adding any other connection to the network, such as from a hidden node to an output. One advantage of this algorithm is that it takes no additional computational time to evaluate features beyond the standard NEAT evolutionary search for better networks. Relevant features are identified based on the fitness of the networks which include them, and high fitness scores cause those networks to propagate, thereby selecting the feature into the population. However, random selection of features leads to the main limitation of this method. In the event that the ratio of relevant to irrelevant features is low, FS-NEAT will likely select mostly irrelevant features to include, harming the quality of the learned policy. Even when relevant features are added to the network, the evolutionary process must work around any previously added irrelevant features, further complicating the NNs and slowing the arrival at a reasonably fit policy. Feature Deselective-NEAT [27] is a similar algorithm which removes connections from an initially fully connected network to reduce the subset size. This approach is best suited to situations where there are many relevant features, and shares FS-NEAT's limitations due to the random search. Incremental Feature Selection Embedded in NEAT has also been recently explored [28]. This method performs a sequential forward search and

evaluates a feature subset by performing evolution for a few generations. This principled feature selection method is successful at finding a good feature subset, however it is sample inefficient, costing a multiple of n more samples than the method we propose in this work.

3.5.2 Sample Aware Embedded Feature Selection

Algorithm Description Designing a feature selection algorithm requires that both a search strategy and an evaluation measure be specified. First, let us consider the search component. For a problem with n features, the number of candidate subsets is 2^n, so for any non-trivial problem, an efficient search heuristic must be employed to efficiently manage this exponential space. Here we adopt sequential forward search (SFS), an efficient search strategy which has been widely used in supervised feature selection [29]. Starting from an empty set, SFS iteratively adds one feature at a time to the current best set until a desired number of features are selected or some other termination criterion is met. Although SFS does not guarantee the optimal solution, it is capable of selecting relevant features while keeping irrelevant or redundant features out of the final subset, given an accurate relevance measure. Additionally, SFS is consistent with the complexification principle underlying the NEAT algorithm. While traditional NEAT only adds hidden layer topological structure over time, SFS adds nodes to the input layer.

An evaluation measure is needed to differentiate between the quality of two candidate feature subsets. Given the choice of NEAT as a learning algorithm, a simple approach would be to use a wrapper strategy to tailor a feature set to NEAT, similar to what was done for IFSE-NEAT [28]. Following the SFS search, if we have previously selected k features, choosing feature $k+1$ amounts to initializing $n-k$ NEAT instances, each pairing a different candidate feature with the selected subset. The subset that causes the highest performance gain after all instances terminate is the new selected subset. This process repeats until the subset search finishes, resulting in $O(n^2)$ runs of the NEAT algorithm, an expensive task in terms of both computational and sample complexity. Instead of relying on the performance of the algorithm, we propose a Sample Aware Feature Selection evaluation measure embedded into the NEAT process (SAFS-NEAT), eliminating the need for *any* additional runs of the NEAT algorithm to perform feature selection. We provide details of the measure later in this section.

These components are encapsulated in Algorithm 5. Prior to selecting the first feature, a data set D is needed. Such a data set can be provided from pre-existing data or by other means, but if unavailable a suitable initial D can be constructed by running NEAT for a single generation and using the samples generated by the champion. The population is then initialized to have no inputs connected before starting the main loop of the algorithm. The outer loop identifies the best feature S_i given the current selected subset and D (see Algorithm 6), and then combines S_i with the population. This is done by adding zero weight connections from input S_i directly to all output nodes. The connection weights are set to zero to preserve the prior policies in the network and allow evolution to determine how much the new feature should affect the existing structure. This mechanism allows feature selection to be embedded into the NEAT algorithm without resetting learning whenever the feature set is modified. The inner loop causes NEAT to use the current selected feature subset until a stagnation criteria is met (discussed shortly). Termination occurs when NEAT converges on a worse policy with the new feature subset than with the previous subset. Other stopping criteria can be integrated, such as a target fitness or a cap on evolutionary generations.

Stagnation plays a key role in Algorithm 5. It both directs when feature selection is to occur, and can bound the complexity of the algorithm. Instead of letting NEAT evolve for a large number of generations, stagnation directs evolution to stop as soon as the fitness gains diminish, indicating

Algorithm 5 IFSE-NEAT(w, ϵ)

```
1: //w: stagnation window size
2: //ϵ: stagnation value threshold
3: S_act ← EMPTY //initialize the selected feature subset
4: population ← INIT-POP(fully-connected-network)
5: D ← NEAT-EVOLVE(population)
6: best ← BEST-QUALITY(population)
7: population ← INIT-POP(no-connected-features)
8: //evolve and add features until fitness decreases
9: repeat
10:    S_i ← BEST-FEATURE(S_act, D)  //see Algorithm 6
11:    Add S_i to S_act
12:    //add current best feature S_i to population
13:    population ← COMBINE(S_i, population)
14:    //build stack of the 2w recent champions for stagnation check
15:    local_champions ← EMPTY
16:    //evolve population using NEAT until stagnation
17:    repeat
18:       D ← NEAT-EVOLVE(population)
19:       //store the champion from population
20:       Push BEST-QUALITY(population) on local_champions
21:    until STAGNANT(local_champions, w, ϵ)
22:    prev_best ← best
23:    best ← Pop from local_champions
24: until best.fitness < prev_best.fitness
25: return prev_best
```

that learning is near-complete with the current feature set, and better gains might be found by expanding the subset. Stagnation requires a window size parameter w, and a fitness improvement threshold ϵ. Since fitness improvements can be noisy, stagnation measures the average fitness of the champion network over one w duration and compares it to the average over the following w generations. If the difference in fitness across this $2w$ period is less than ϵ then learning is considered to be stagnant, and the inner loop is stopped. In practice, both parameters are easily set: w to a small number of generations (i.e., 5-15), and ϵ to a small fraction of the maximum attainable fitness for the problem.

Evaluation Measure Collecting samples during the NEAT evolutionary process yields a data set $D = \{\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s'}_i\}_{i=1}^{d}$, one tuple for each of d interactions with the environment. The components $\mathbf{s}$ and $\mathbf{s}'$ are vectors of length n representing the current and next state feature values, respectively, r is the immediate reward, and $\mathbf{a}$ is a vector of length m containing each of m action decisions that were simultaneously made in state $\mathbf{s}$. Note that the environment reports the value of each feature S_i whether or not S_i is in the selected subset. With this data set, the ability to use supervised filter feature selection techniques arises if a class label can be assigned. Two recent feature selection algorithms for RL have addressed this difficulty by using r as the class label [24, 30]. This designation worked well in both of those scenarios because samples were gathered from known good policies. During learning, however, each feature may have only a weak correlation to the reward value since the current π may be distant from π^*. This problem is magnified in the common cases where reward information is delayed or sparse. Instead, we choose to study the relationship between the feature values of $\mathbf{s}$, $\mathbf{s}'$, and $\mathbf{a}$. We seek to measure the dependency in the change of each state value ($s'_i - s_i = \Delta s_i$) given the action a_j taken at state s_i for each element in the collected data set D.

There are various dependency measures in the feature selection literature [31, 16]. Here we use

Algorithm 6 BEST-FEATURE($\mathcal{S}_{act}$, D)

```
1: //S_act: set of currently selected features
2: //D: data set of collected samples from environment
3: S_best ← S_0
4: best-score ← −1
5: for i = 1 → (n − |S_act|) do
6:     relevance ← 0
7:     for j = 1 → m do
8:         relevance ← relevance + I_D(ΔS_i, A_j)    (using Eq.(1))
9:     end for
10:    redundancy ← 0
11:    for S_k ∈ S_act do
12:        redundancy ← redundancy + I_D(S_i, S_k)
13:    end for
14:    //keep track of best feature and its score
15:    if best-score < relevance/redundancy then
16:        best-score ← relevance/redundancy
17:        S_best ← S_i
18:    end if
19: end for
20: return S_best
```

mutual information

$$I_D(X,Y) = \int_Y \int_X p(x,y) \log \left(\frac{p(x,y)}{p(x)p(y)} \right) dx\ dy, \tag{1}$$

which measures the dependency between two continuous random variables X and Y based on some sample data D. The higher $I_D(X,Y)$ is between the two variables, the more related the distributions of X and Y are. A value of zero from the measure indicates that the variables are independently distributed. Here we are interested in $I_D(\Delta S_i, A_j)$ for each $S_i, 1 \leq i \leq n$ and $A_j, 1 \leq j \leq m$. Intuitively, features with values that appear to change independently with respect to any action selection are of little obvious use to the policy and would be given low scores. This measure is equally applicable for discrete variables by exchanging integrals with summations in Eq.(1). More research is required in the area of filter feature selection for RL to determine the best relevance measurement technique for this type of data, however, we note that mutual information is intuitively appealing and works well empirically.

The complete evaluation measure is expressed in Algorithm 6. This algorithm follows the minimal-redundancy-maximal-relevance (mRMR) strategy to rank features [32]. Calculating relevance of a feature S_i requires summing the pairwise mutual information (as computed in Eq.(1)) between ΔS_i and each of the m action variables to capture how a feature responds to each individual action available. The redundancy measure is similarly carried out by summing the mutual information between the candidate feature and each feature that has already been selected. The score of the feature is the quotient of its relevance score over its redundancy score, and the feature with the highest score is returned from the function. Note that this computation needs only to be done for features which are not currently included in the subset. Computing mutual information is linear in the number of samples, and this cost is dominated in practice by the cost of running the NEAT algorithm.

4 RESULTS AND DISCUSSION

We have thoroughly evaluated and analyzed our approaches to state space abstraction and feature selection. We report on these empirical results here.

4.1 Pertaining to Evolutionary Tile Coding

4.1.1 Experimental Setup

We conducted an empirical comparison of CMAC, ATC, and EvoTC on two well known RL benchmark problems with continuous state spaces. The purpose of these algorithms is to reduce the size and complexity of domains' state spaces and enable a RL algorithm to discover an optimal policy for problems in those domains. We measure the effectiveness of the approaches by the number of states in the abstract state space and by the number of learning updates required by the RL algorithm to learn an optimal policy. The fewer the number of states in abstract state space translates to the method's ability to more effectively abstract the state space. And, the fewer the number of updates required by the RL algorithm to learn an optimal policy, the better the state abstraction.

The following is a description of the benchmark problems used and our experimental setup. It should be noted that all the methods require some parameter tweaking in order to achieve their best performance. In our comparisons we used the best found parameter settings for each method. The parameters used for each method and problem are specified below. Each method was paired with the RL algorithm SARSA [22] to derive policies. Also, the results shown for EvoTC are representative of the median value of 25 separate runs. Because EvoTC is dependent on a stochastic search, several runs with different random seeds were necessary to properly characterize its performance.

Mountain Car The mountain car problem is a classical control RL problem in which the learner has to derive a policy to enable an automobile to escape a deep valley. The car does not have enough power to drive up the sides of the valley starting from a standing position. To get out the driver must build up enough momentum by rocking back and forth. Two continuous features, position and velocity, specify the state. At each time step the RL algorithm has to select one of three possible actions; accelerate to the left or right, or coast. A reward signal of -1 for every time step the car has not reached the goal state is provided to encourage the discovery of a policy that reaches the goal state in as few time steps as possible.

We use a problem set of 100 different starting positions and initial velocities to represent the problem domain in our experiments. The algorithms are evaluated based on the average performance over all instances in the problem set. For our problem set an optimal policy enables the car to escape the valley in average of 50 time steps.

In our experiments for CMAC we used 2 layers of tiling with 11 tiles per feature for each layer. This allows a maximum of 242 possible unique abstract states. ATC requires the *split threshold* parameter be specified. For the mountain car problem we found a value of 521 to work well. EvoTC requires the mutation probabilities be specified. For this problem values of 32% for *shift* and 5% for *divide* per tiling per generation were used. A population size of 100 was also used for each evolutionary generation.

Pole Balance The pole balance problem models a car balancing a long pole attached on a hinge [33]. The car is free to travel on a short track to keep the pole balance vertically over the car. Failure occurs if the pole falls more than 12 degrees from vertical or if the car rolls off either end of

Table 1: Results for mountain car

	Number of Updates	Number of States
CMAC	1.22e+05	177
ATC	1.88e+05	83
EvoTC	2.00e+07	2

the short track. The state is represented by 4 continuous features; the position and velocity of the car, and the angle and angular velocity of the pole. There are three available actions; accelerate to the left, to the right, and to coast.

We use a problem set of 20 different initial feature values in our experiments. The goal for the algorithms to find a policy that keeps the pole balanced for at least 10^6 updates without dropping the pole or exceeding the bounds of the track.

For CMAC, the settings of 2 layers of tilings with 11 tiles per dimension of input per layer is again selected for this test for a maximum of 29282 states. The settings selected for EvoTC are 30% for *shift* and 12% for *divide*. We were unable to successfully apply ATC to this problem.

4.1.2 Results and Discussion

The results of the mountain car and pole balance are listed in Table 1. All three methods were able to converge to an optimal policy. We can see that CMAC was able to solve the mountain car problem in the fewest number of updates. This is slightly surprising because it was shown that ATC was able to outperform CMAC on this problem in [3]. We were not able to reproduce that result[1]. This result is intuitive however, in that the fixed CMAC tile coding was tuned for this problem and was found as a result of many trial runs. ATC and EvoTC have to learn their tile abstractions and this requires some additional time and updates.

It should be noted that EvoTC is penalized by the update metric because all the updates required by the failed members of the population are included. Including the aggregate updates required for all the members of the population is necessary to get an accurate measure of computation time required. However, each evaluation of a tiling per generation could be done independently in parallel, which would result in a significant speed up of this algorithm.

Table 1 also shows the size of the abstract state space required for each method. CMAC only uses 177 of the potential 242 states available. EvoTC and ATC are able to solve the mountain car using substantially smaller state spaces which shows they derive much more efficient state abstractions. This suggests that they will be able to scale more effectively as the size of the state spaces increase.

The most striking result of this experiment is that EvoTC was able to derive an optimal policy using an abstract state space consisting of only two states. EvoTC was consistently able to find this state abstraction during our experimentation. The mountain car problem is one of the classic RL control problems. It is considered difficult due to its continuous state space. EvoTC simplified it to a simple two state problem which is trivial for a RL algorithm to find a policy for. Not only that, EvoTC was able to eliminate the need for an entire feature. The only split in the state space occurs at .477 of the velocity vector. There are no divisions over the position feature which means it is not relevant at all to solving the problem. This result highlights the power of automated state abstraction to find unintuitive and effective abstractions.

[1]The authors of the ATC work were contacted and informed of this.

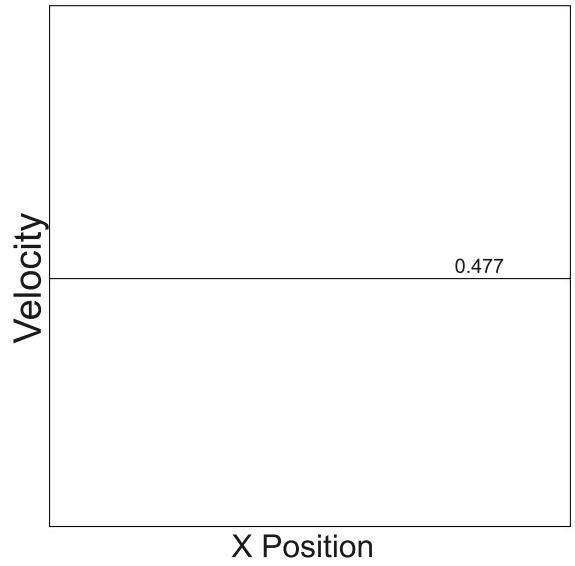


Figure 5: This figure shows how EvoTC algorithm discretized the mountain car state space

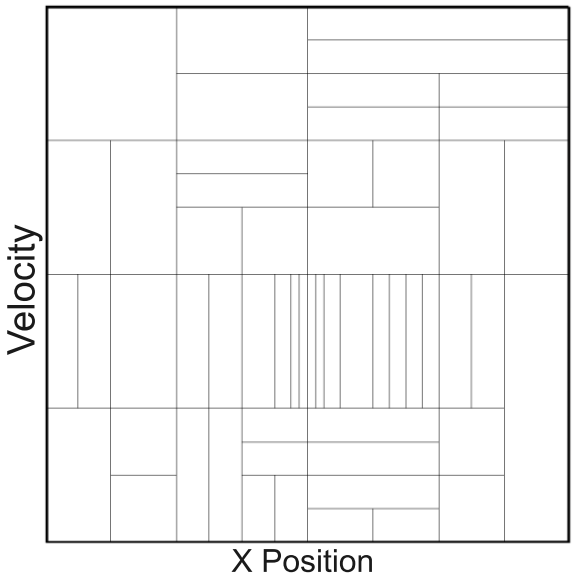


Figure 6: This figure shows how the ATC algorithm discretized the mountain car state space

Table 2: Results for Pole Balance

	Number of Updates	Number of States
CMAC	3.69e+08	5379
ATC	failed to converge	failed to converge
EvoTC	1.04e+09	61

This experiment also shows how important the design of the abstraction can be. Figures 5 and 6 shows the abstract state spaces derived by EvoTC and ATC respectively. The abstraction derived by ATC is significantly more complex than the one derived by EvoTC and includes divisions across the position vector. ATC cannot find the same abstraction that EvoTC is able to find because it arbitrarily divides each tile evenly. As a result it had to derive a much more complex abstract state space to learn an equivalent policy.

Table 2 shows the results we obtained applying these methods to the pole balance problem. The pole balance problem is significantly more difficult than the mountain car problem in that it has double the number of continuous features. As such, we can see that CMAC still requires the fewest updates, but required significantly more abstract states in order to solve this problem. In our experiments we were unable to find a parameter setting that enable ATC to converge. Once again EvoTC was able to derive an abstraction with far fewer states and still allows the RL algorithm to find an optimal policy. EvoTC still required an order of magnitude more updates than CMAC, however the increase in number of updates and states required by CMAC compared to EvoTC further implies that EvoTC will scale more effectively as the size of the state space is increased.

In our testing we found that all methods were extremely sensitive to untuned parameter settings. Slight changes to the parameter settings that work for a domain could very easily prevent these methods from converging again. This was especially true of the ATC algorithm, which required a substantial amount of trial and error to find a parameter setting that worked for the mountain car. Finding settings for CMAC and EvoTC was significantly less time consuming, but still required some trial and error.

Although EvoTC and CMAC were able to solve both benchmark problems it does not appear that either method will scale adequately as the number of features that describe the state space is increased. Both methods are tile coding based and are linear abstractions of the state space. As a result, although the abstract state spaces found by these methods are significantly smaller than the actual state space, they will still scale proportionally as the number of features is increased. It may be the case that non-linear state abstraction methods such as RL-SANE [10] are necessary as the number of features are increased.

4.2 Pertaining to Continuous State Space Abstraction

4.2.1 Results and Discussion

Table 3 shows the average fitness over each of the problem sets for the mountain car and double pole balance domains. The mountain car problem shows all five methods performing very similarly and all rapidly converging to a policy that takes on average approximately 50 time steps to navigate the car from the valley. Both of the more sophisticated methods, MDS and TRE, lag behind the top performers somewhat, which indicates that this problem can be easily learned without complicated abstract state repartitioning. These results do serve to show that using automatically repartitioning of the abstract state space does not hurt the overall convergence of the learner

	Mountain Car	Double Pole Balance
MDS	3.36 ± 1.25	13.5 ± 5.93
TRE	63.64 ± 23.57	58.13 ± 13.48
Large	14.77 ± 8.6	24.51 ± 15.59
Small	14.82 ± 8.7	24.48 ± 15.43
Fixed	50	10

Table 3: Average number of final abstract states ± stdev and optimal number of states for the fixed abstraction.

on simple reinforcement learning problems even when the problem is simple enough that a fixed abstraction is sufficient.

Examining the fitness curves of the double pole balance problem shows several trends. The most obvious conclusion that can be drawn is that the automatic methods are all able to converge towards the optimal policy at a greater rate than the fixed RL-SANE algorithm. If the number of abstract states are tuned, the fixed RL-SANE method can find the optimal policy at a similar rate as the other algorithms, however, if a range of possible good parameters are used instead the algorithm does not do nearly as well. On the contrary, the mutation methods, both small and large, are able to overcome the arbitrary initial parameters and efficiently repartition the abstract state space to allow the learner to quickly converge to the optimal policy. The MDS and TRE methods started out near the fixed method but rapidly improved to the mutation methods. Towards the end of the reported generations the MDS method shows the best performance overall, validating the idea that allowing a more specialized partition of the state space can lead to improved convergence properties of the learner. TRE proves to be an able abstraction method as well, and the performance of that algorithm is noteworthy early on in the learning process. We can see that it experiences an almost immediate jump in fitness, which may be due to its heuristic which favors separating those observations which may be able to reach previously unexplored areas of the state space if they are able to follow actions that are not preferred by other nearby observations.

Table 3 contains the average final number of abstract states for each automatic abstraction method as well as optimal number of states for the fixed tiling. We can immediately see that the optimal number of states for the fixed RL-SANE algorithm is not the number of states that each of the automatic methods tend to; only TRE on mountain car and MDS on double pole balance are similar. TRE tends to break up the space into many more states than the other methods, while MDS leads the abstraction towards fewer states. This implies that there are relatively few clusters of observations in the abstract space, but there are many repetitive substructures in these clusters when the order of observations are considered. Both of the mutation methods converge to similar low numbers of states in the final abstractions, which explains why their fitness measures in Table 3 are also very similar.

4.3 Pertaining to Automatic State Space Abstraction

4.3.1 Experimental Setup

Here we see what benefit MDS gives the RL-SANE algorithm in terms of convergence speed and number of abstract states in the solution. In addition to comparing the automatic MDS method against the fixed tiling of RL-SANE, we include another algorithm in the study, a mutation method which allows RL-SANE to mutate the number of abstract states during the evolution of the network.

The experiments are carried out on two benchmark RL problems, mountain car and double pole balance.

Only two perceptions are used to define this problem, the position of the car within the valley X, and the velocity of the car V. Time is discretized into small intervals and the learner can choose one of two actions in each time step: drive forward or backward. The only reward that is assigned is -1 for each each action that is taken before the car reaches the goal of escaping the valley. Since RL algorithms seek to maximize the reward, the optimal policy is the one that enables the car to escape the valley as quickly as possible, and therefore receive as few negative rewards as necessary.

This is a higher dimensional problem than the mountain car problem, with six perceptions being given to the learner: the position of the cart X, the velocity of the cart X', the angle each beam makes with the cart, θ_1 and θ_2, and the angular velocities of the beams, θ_1' and θ_2'. Once again, time is discretized into small intervals, and during any such interval the learner can choose to push the cart to the left or right or to leave it alone. In our experiment, the learner only receives a -1 reward for dropping either beam or exceeding the bounds of the track. If the learner is able to balance to poles and not exceed the bounds of the track for 10^6 time steps the problem is taken to be solved.

On each of the problems the three methods were evaluated over 25 runs using different random seeds (the same seed values were used for all three methods). For each run, both the mountain car and double pole balance environments used a problem set size of 100 random initial start states. We report the average values across the 25 runs in our results. It should be noted that the mutation and the fixed tiling approaches have a significant dependency on the initial number of abstract states, while the MDS does not. In the mutation and fixed methods we experimented with setting the number of initial abstract states from $10, 20, \ldots, 100$ and the results show either the average performance of the algorithm over all of these boundaries, or the best performer from the 10, as indicated. For the RL-SANE algorithm with a fixed abstraction, these initial states cannot change during the learning process, while the mutation method is free to alter them over time. The MDS method begins with an arbitrary abstraction over the state space which is quickly replaced by a more competent estimate after the first attempt at learning the problem. Prior experiments have shown that the fixed RL-SANE algorithm achieves the best learning rate with 50 abstract states in mountain car and 10 in double pole balance, both of which are included in the abstract state ranges that were tested on.

The RL-SANE algorithm was set to use a population of 100 neural networks per generation, with a maximum of 200 generations of learning. Neuroevolution is provided by Another NEAT Java Implementation (ANJI) [2]. We used the Sarsa(λ) learning algorithm with learning and neuroevolution parameters set as in [10], and we also limited each learning episode of the mountain car to 2500 time steps to ensure termination. The mutation method was allowed to alter the number of states by up to 5 per generation to provide regularity between generations of neural networks. For MDS the density of the observations in the state space was estimated using a histogram of 1000 evenly spaced bins to collect observations. The exact value of this parameter is unimportant as long as it is significantly larger than the number of expected abstract states in the solution and an episode produces enough observations to partially fill in the space.

4.3.2 Results and Discussion

Presentation and discussion of the results is broken up in two parts based on problem domain. The first section addresses the mountain car problem and the double pole balance problem is analyzed in the second.

[2]Source code available at `http://anji.sourceforge.net`

Table 4: Average number of final abstract states used ± standard deviation for MDS and the number of initial states used to derive the best performance for the mutation and fixed approaches.

	Mountain Car	**Double Pole**
MDS	3.36 ± 1.25	13.5 ± 5.93
Mutation	90	10
Fixed	50	10

Mountain Car Figure 7 shows the average number of time steps taken to leave the valley over the 25 runs for the three methods. Fewer steps are better. The fixed and mutation curves shown use the best choice of initial number of abstract states, although all values attempted gave similar results for this problem and so are omitted for clarity. The mutation method found the optimal policy fastest with 90 initial states, and the fixed approach did the best on 50 states. The mountain car problem shows all of the methods performing very similarly and all rapidly converging to a policy that takes on average approximately 50 time steps to navigate the car from the valley. The Maximum Density Separation method lags slightly behind the other two methods, which indicates that this problem can be easily learned without complicated abstract state repartitioning. Analysis of Table 4 gives an additional explanation for the performance of MDS. The MDS method consistently finds that there are only roughly three groups of observations in the space and so only partitions the space into three tiles. It might have taken additional generations to effectively learn the correct placement of these partitions, compared with the larger number of states used by the mutation and fixed methods. Even though the other methods need to learn correct values over more states, there are still relatively few states to learn so they quickly converge. These results do serve to show that automatic repartitioning of the abstract state space does not cause too much degradation to the overall convergence of the learner even when fixed tilings can work well. The overall convergence of the learner on simple reinforcement learning problems even when the problem is simple enough that a fixed abstraction is sufficient.

Table 4 contains the number of abstract states available to the fixed and mutation methods during their best run, as well as the number of states that was determined by the MDS method. We can see that the MDS method on average uses 3 or 4 abstract states with a standard deviation of 1.25, meaning that it was very consistent in the number of states used to learn the problem. The other two methods preferred many more partitions. However, the number of those partitions where observations were placed by the ANNs in the RL-SANE procedure was much smaller. The fixed method used roughly 19.08 ± 9.08 (average±std) out of the 50 available states, while the Mutation method used 30.40 ± 22.67 from 90 possible starting states. From this we can see that it is not only the number of tiles that is important, but the effect they have on where the tiles get placed across the abstract state space. Since each of these two methods uses fixed-width partitioning, the number of abstract states will cause the boundaries to fall in different locations, and over-partitioning the space can allow the learner to use more appropriately positioned tiles. This is in contrast to the MDS approach which allows the boundaries to be placed anywhere in the abstract state and does not need to add additional empty states solely to adjust the layout of the useful ones.

The disparity in the number of used states between three methods is interesting. Two possible explanations regarding why the fixed and mutation methods had a large number of states are that the large number of initial states induced the learner and ANN to prefer many small groups of observations throughout the one dimensional state space, or that many adjacent states shared the same optimal action preference. In the latter case, these adjacent states could have been merged

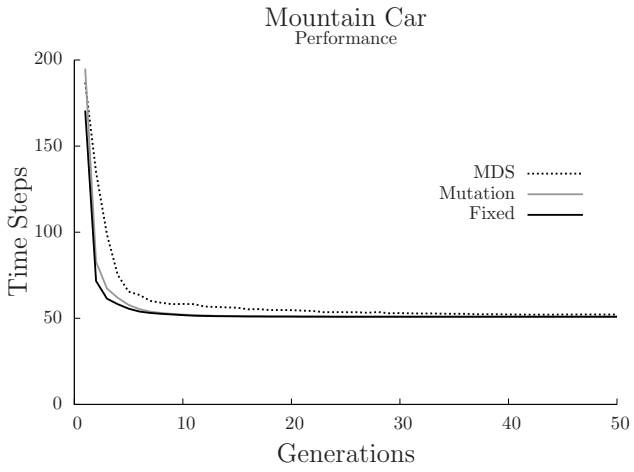


Figure 7: The performance of MDS, mutation and the fixed abstraction methods on the mountain car. The curves for the mutation and fixed methods are the results from the best initial parameter settings.

together resulting in possibly far fewer states, though there is no means to do this in these methods.

Double Pole Balance Figure 8 shows the average number of time steps the pole was balanced for each of the three methods; the solid lines for the mutation and fixed methods are the best scores achieved by any initial parameter setting (10 for both methods). The more time steps, the better the algorithm has learned the problem. The dashed curves show the average performance for the two methods across all tested initial state boundaries. The MDS has no initial parameter selection and so only has the single dotted line in the figure. We can see that MDS and the best settings of the other methods show similar trends, with the score of the two adaptive methods just edging out the best fixed method. Analysis of the average curves (dashed lines) gives more information about the general performance of the mutation and fixed methods as compared to the MDS method. While the MDS method has no choice of initial parameters and still ends up achieving an excellent overall score, different numbers of initial abstract states causes a varied performance in the other two methods. The mutation method is relatively robust with regard to the initial parameter selection as compared to the performance degradation seen by the fixed method if a bad initial abstraction is selected.

Figure 9 explores this phenomenon more completely by showing the performance of the fixed and mutation methods for all 10 initial parameter settings. The individual parameter results are shown in faint gray lines except for the best and worst performers which are solid black lines. The vertical bars span the space between the best and worst performer and highlight the sensitivity of the fixed method. Generally, the smaller parameters perform better than the larger initial values, and the vertical bars show that the fluctuation of performance is much smaller in the mutation method than the fixed method. The reason why the double pole balance problem is much more sensitive to the initial number of abstract states as compared to the mountain car problem has to do with the number of actively used states. In the mountain car problem, even if many states were available only a fraction of those were used. This is in contrast to the double pole balance problem where nearly all of the available states are used. The mutation method allows the number of available states to quickly be reduced down to a number that the learning algorithm can deal with, and thus improve the rate of convergence compared to the fixed method.

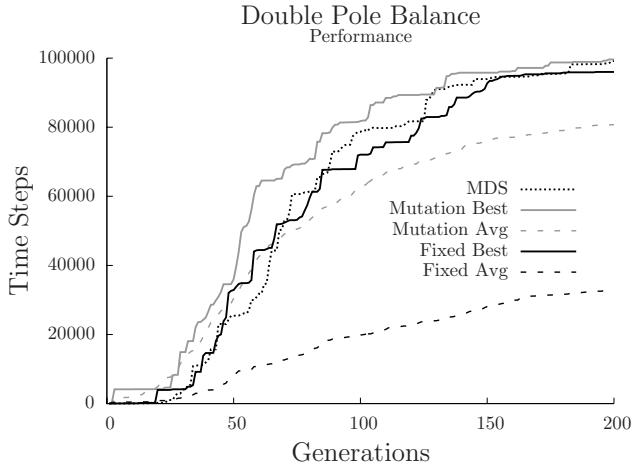


Figure 8: Learning performance evaluation. For comparison, the mutation and fixed methods include their single best initial parameter setting as well as their average performance across all parameter values.

All three abstraction methods prefer to use a similar number of abstract states for the double pole balance problem, as indicated by Table 4. The best fixed tiling began with 10 tiles and used all 10 consistently throughout the learning process. The mutation method also achieved its best performance after starting with 10 tiles, however the number of tiles used decreased to 7.56 ± 3.88 by generation 200. The MDS method was also in agreement with the other approaches and by generation 200 was using 13.5 ± 5.93 tiles. Despite the complexity of the problem, many of the states in the ground state space can be successfully aggregated together, as evidenced by the small number of states being used.

4.4 Pertaining to Incremental Feature Selection

We analyze the performance of our IFSE-NEAT algorithm from two perspectives: (i) the quality of the derived policy, and (ii) the ability of the algorithm to select relevant features. We measure the quality of the derived policy by a problem-specific *fitness function*. The composition of the selected subset in terms of the *fraction of relevant features among selected ones* quantifies an algorithm's ability to select a good feature subset. Finally, we verify that the performance of our algorithm (measured by the above metrics) does not degrade as the number of irrelevant features increases.

We compare IFSE-NEAT to the baseline NEAT as well as FS-NEAT, a competing feature selection algorithm we describe in Section 4.4.1. All three algorithms are evaluated in a challenging race track domain that is capable of providing many relevant and irrelevant features for the algorithms to work with. The details of this environment as well as the specific parameters used by the algorithms are given in Section 4.4.2.

4.4.1 FS-NEAT

Feature Selective NEAT, or FS-NEAT, is an embedded feature selection algorithm within the NEAT framework [20]. One limiting assumption standard NEAT makes, discussed in Section 3.4.2, is that all input features are relevant and are fully incorporated into all solution networks. FS-NEAT assumes that few features are actually relevant. Networks are initialized with only a single

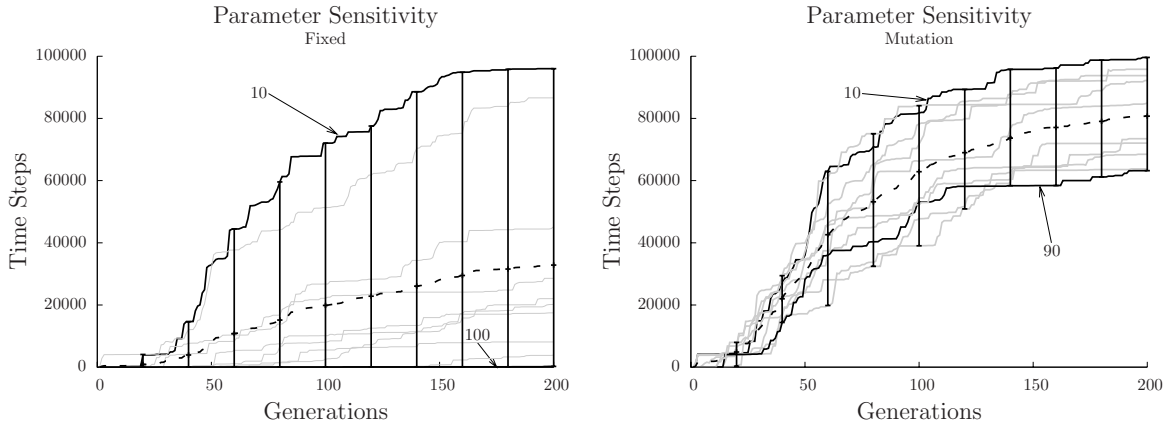


Figure 9: The effect of different numbers of initial abstract states for both the Fixed and Mutation methods on the Double Pole Balance problem. The dashed line is the average of all values; vertical bars show distance between maximum and minimum performance.

Table 5: Different problems used in the RARS experiments broken down by the number of relevant, irrelevant, and total number of features.

Relevant	**Irrelevant**	**Total**
5	5	10
5	25	30
5	45	50
5	95	100

connection between a randomly selected pair of input and output nodes. Through subsequent mutations other input nodes may add a connection to the rest of the network and hence be selected into the model.

4.4.2 The RARS Domain

We conducted our experimental analysis using version 0.91_2 of the Robot Auto Racing Simulator (RARS)[3]. RARS provides a detailed physical simulation of a racetrack and vehicles and allows users to define their own artificial agents to control the racers.

The goal of the simulation is to learn a path around the track that covers the most distance in a limited time while minimizing damage received by the car. Damage is calculated by RARS based on the amount of time the car spends off the track. The racers are controlled by supplying a desired speed and direction at every time step in the simulation.

We implemented a rangefinder system in the simulation to provide vehicle position information to the learning algorithm as in Figure 10. In our experiments we placed N range sensors evenly around the front of the car as in [20], starting from the left side of the car and finishing at the right to provide a full view of the track. The range finders, together with the velocity of the car, are used by the learner to provide two continuous control outputs, corresponding to the desired speed and direction of the car.

[3] `http://rars.sourceforge.net/`

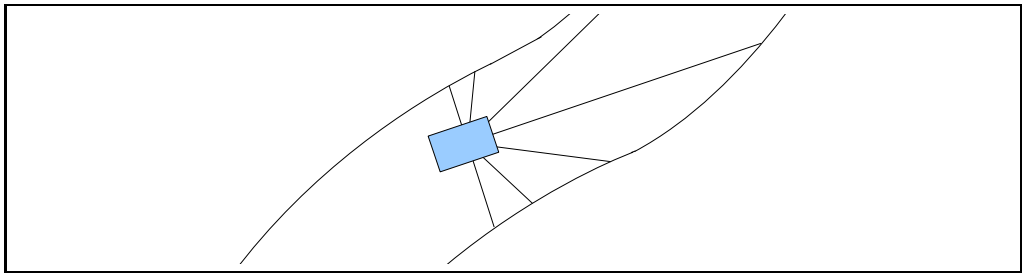


Figure 10: Providing vehicle location information via a rangefinder system.

To make the RARS environment challenging from a feature selection point of view we added *irrelevant* features to the set. Irrelevant features simply return a random value in [-1,1]. We developed several challenging problems with different combinations of relevant and irrelevant features as shown in Table 5. These combinations allow us to examine the robustness of each of the three algorithms in comparison w.r.t. increasing numbers of irrelevant features.

All three algorithms tested are neuroevolutionary algorithms that require a fitness function to provide the feedback that guides learning. We adopt the fitness function used by [20], $S = 2d - r$, where d is the distance the car has traveled from the start and r is the amount of damage received. Trials end after the learner either has observed 2000 time steps or the car registers too much damage.

All experiments took place on the clkwis.trk track that is bundled in the RARS package, shown in Figure 11. This track was selected because it exhibits several driving scenarios such as straightaways, turns and an S-curve. The experiments were conducted in the RARS environment according to the following setup.

- Three algorithms were tested, NEAT, FS-NEAT, and IFSE-NEAT
- For each tested combination of features 10 runs were conducted with each algorithm, results presented are the **average** of these 10 runs.
- Each run lasted 200 generations
- IFSE-NEAT split the 200 generations into five $L(\cdot)$ periods with $L(1) = 3,\ L(2) = 7, L(3) = 20,\ L(4) = 50,\ L(5) = 120$
- The NEAT population size was set to 100

We set the number of generations allowed to 200 since the algorithms appeared to converge by that point and there was no need to carry the experiment further. The particular values of the $L(\cdot)$ function are not important, and we experimented with other values which yielded similar results.

All three algorithms in comparison rely on NEAT for generation of the neural networks to allow learning. In our experiments we make use of Another NEAT Java Implementation (ANJI) for the NEAT algorithm [4]. We followed the settings given in [20] of 0.10 and 0.02 for add-connection and add-neuron respectively to set the parameters for the FS-NEAT algorithm. For NEAT, and IFSE-NEAT we set the add-connection mutation probability to 0.02 and the add-neuron mutation to 0.01. In our experiments we found the parameters used with FS-NEAT to be too aggressive for NEAT and IFSE-NEAT.

[4]Source code available at `http://anji.sourceforge.net`

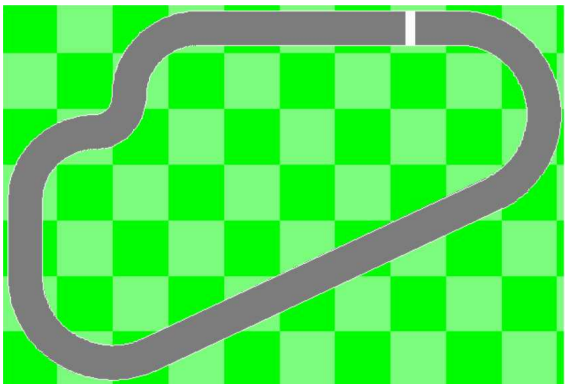


Figure 11: A top-down view of the clkwis track used in the experiments.

4.4.3 Results and Discussion

Figure 12a shows the results of running the three algorithms for the problem with 5 relevant features and 45 irrelevant features as the algorithms search for the optimal policy. Fitness of the derived policies is measured in terms of the value of the fitness function defined in Section 4.4.2. We can see that IFSE-NEAT converges to the best fitness of the three algorithms, and this convergence occurs at around generation 50. Both NEAT and FS-NEAT exhibit a slower rate of convergence than IFSE-NEAT. In this situation, NEAT is limited by the need to assign correct weights to many features. Since all available features are used in the NEAT neural network, NEAT has to evolve through many generations to find the right weights for links associated with the relevant features while keeping weights for irrelevant features low in order to limit their impact on the network output. The FS-NEAT algorithm suffers from its random search policy. Since there are many irrelevant features in the problem, they have a higher chance of being included in the network than a relevant feature does, causing the algorithm to be slow to learn an effective policy.

In Figure 12b we see the composition of the selected subsets by the three algorithms. IFSE-NEAT clearly has the highest percentage of relevant features per selected group, at around 90% on average. This number begins at 100% for 1 selected feature and slowly decreases as features are added to the set. Figure 12a shows that IFSE-NEAT achieves optimal fitness early, and then even relevant features do not appear helpful, causing some irrelevant features to be incorrectly selected in some of the 10 runs of the algorithm. FS-NEAT slowly adds new features to the set, many of which are irrelevant, causing low scores in both measures. It should be noted that IFSE-NEAT and FS-NEAT select around 5 features by the 200^{th} generation in all tested settings.

We now further study how the three algorithms scale with an increasing number of irrelevant features. Figure 13b shows the fraction of relevant features among the selected ones by each algorithm. We can see that for each of the problems, IFSE-NEAT selects on average at least four relevant features in five feature selection steps. This validates our feature ranking and selection criteria, and supports the consistently good fitness values seen in Figure 13a. As predicted, NEAT's fitness degrades as the number of irrelevant features increases and the fraction of relevant features decreases. It always includes all the irrelevant features, which increases the complexity of the networks and slows down learning. FS-NEAT's fitness shows a variable trend caused by the random selection mechanism. Despite starting with more irrelevant features in the problem with 50 features, the fitness of the final policy actually improved over the problems with 10 and 25 features, as shown in Figure 13a. This is most likely the result of the network weights being randomly improved by chance and more trials should remove this effect.

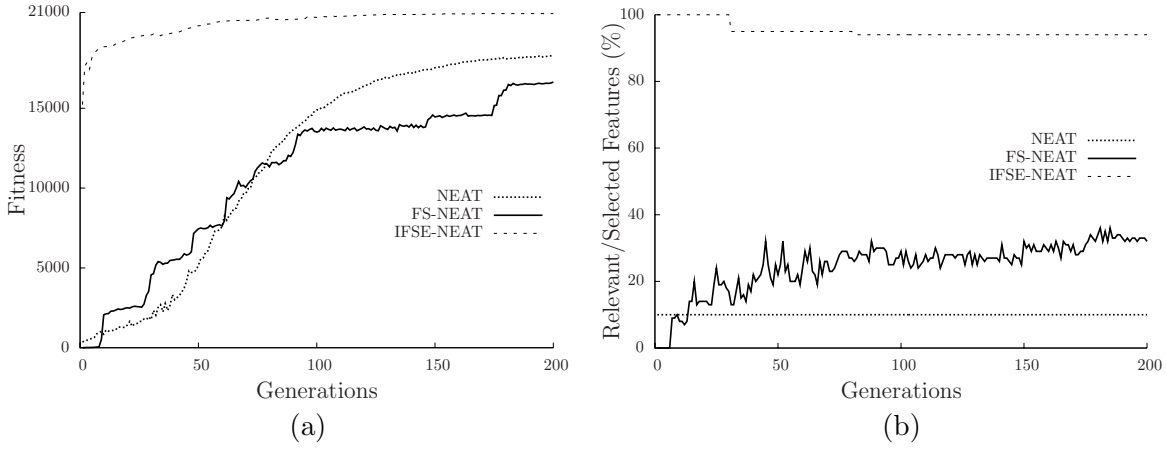


Figure 12: Two performance metrics: fitness (a) and the fraction of relevant features among the selected ones (b), for NEAT, FS-NEAT, and IFSE-NEAT across 200 generations on the problem with **5 relevant** features and **45 irrelevant** features.

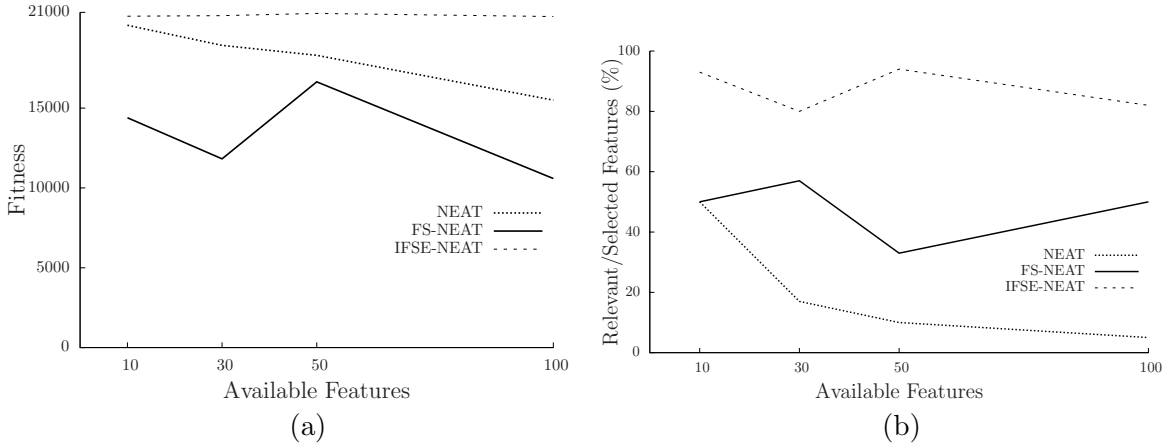


Figure 13: Two performance metrics: fitness (a) and the fraction of relevant features among the selected ones (b), for NEAT, FS-NEAT, and IFSE-NEAT at the 200th generation **across 4 problems** with 5 relevant features and 5, 25, 45, and 95 irrelevant features.

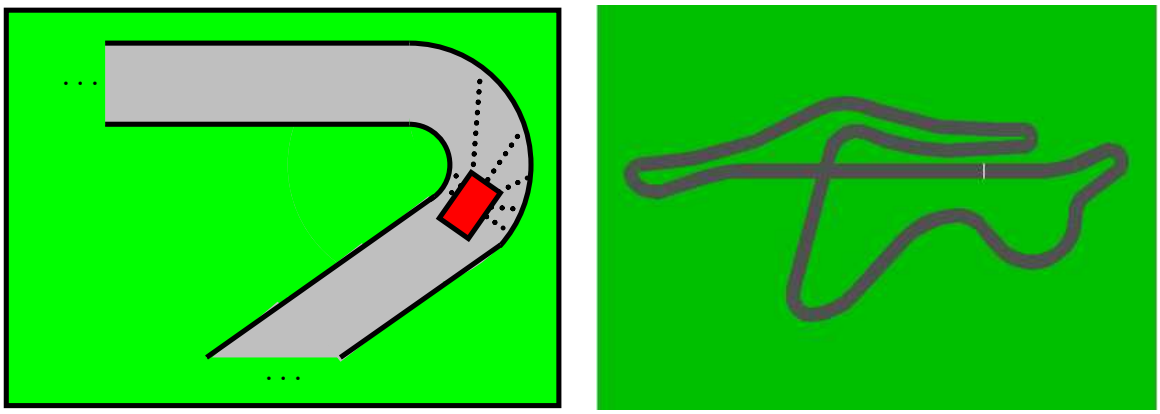
4.5 Pertaining to Sample Aware Feature Selection

This empirical study demonstrates the ability of SAFS-NEAT to scale up RL in high-dimensional environments. Section 4.5.1 details the two domains used in this work, and gives parameter settings for the algorithms in comparison. Results from these domains are provided in Section 4.5.2, along with a discussion of our main findings.

4.5.1 Experimental Setup

Robot Auto Racing Simulator One experimental domain is the Robot Auto Racing Simulator[5] (RARS) racing simulation environment. The goal of this problem is to drive a car around a track as quickly as possible, while keeping the car on the track. Each state of the environment is defined by a set of position sensors and the car's speed. The sensors evenly span the 180° area in front of the car as depicted in Figure 14(a), and measure the distance from the car to the nearest track

[5]Source code available at: `http://rars.sourceforge.net`



(a) (b)

Figure 14: (a) The range finder sensors measuring the car's position relative to the nearest track walls. (b) Overhead view of Fiorano, the track used in this study.

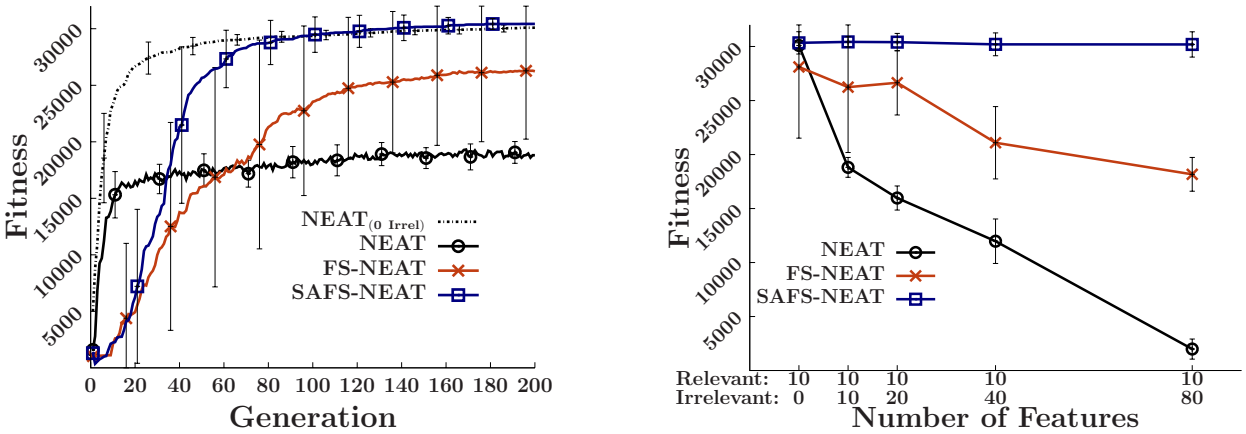
wall along that direction. The set of position sensors and velocity are considered relevant features. In each state, the learner must use these features to determine correct values for two continuous actions, the next desired vehicle speed and direction. Several tracks were experimented with, all results reported here were gathered using Fiorano, a track included in the RARS distribution and depicted in Figure 14(b).

To make the problem more challenging, we introduce irrelevant and redundant features to the environment. Irrelevant features, which may come from sensors for perceptional inputs irrelevant to driving (e.g., radio tuner frequency), are Gaussian random variables with mean 0.5 and standard deviation 0.25. All sensors must report values within the range [0, 1], and any random value that falls out of this range gets clipped to the nearest boundary. Redundant sensors are simply made by adding more range finders to the car, forcing them to be spaced closely together so that neighbors return similar information. Our study investigates the effects of increasing numbers of irrelevant sensors in {0, 10, 20, 40, 80} while keeping the **number of relevant features fixed at 10**. Similarly, different numbers of relevant features {10, 20, 30, 50, 90} are used to create redundant feature scenarios, while keeping the number of irrelevant features set to 0.

The neuroevolutionary algorithms in our experiments require a fitness function to measure the quality of NNs in the population. In the case of RARS, we allow a policy to be executed for up to 3000 time steps in the environment, or until the car crashed, whichever came first. The fitness of a policy is $f(\pi) = 2d - p$, where d is the distance the car has traveled, and p is the internally calculated damage penalty incurred by leaving the track.

Algorithm Parameter Settings Three evolutionary algorithms: NEAT, FS-NEAT, and SAFS-NEAT are studied. All algorithm implementations are based on the ANJI[6] code base developed for the NEAT algorithm. All share a number of parameter settings that control evolution. The population size p was set to 100 in all experiments. The top 20% of the population was propagated unchanged into the next population. Population members are eligible to reproduce if their compatibility scores are greater than 0.5. The compatibility score from [11] was used with weight coefficients: excess = 1.0, disjoint = 1.0, and matching = 0.04. Activation functions on the input neurons are linear, while all other activation functions are sigmoid. Since evolution is a stochastic

[6]Source code available at: `http://anji.sourceforge.net`



(a) 10 Irrelevant Features (b) Irrelevant Scaling

Figure 15: (a) Fitness at each generation during policy learning for NEAT, FS-NEAT, and IFSE-NEAT with 10 irrelevant features. (b) Shows the fitness of each algorithm at generation 200 as irrelevant features increase from 0 to 80. Error bars denote the standard deviation of the 25 runs at each point.

process, all experiments were run 25 times, each with a different random seed, and all reported results are averages across these runs.

For RARS, perceptron networks were found to produce successful policies, so add-neuron-mutation was turned off. The add-connection-mutation was off for NEAT and SAFS-NEAT, while 0.01 was found to be a good value for FS-NEAT. For DPB add-connection was set to 0.02 and add-neuron to 0.01 for all algorithms. Evolution was permitted for 200 generations in RARS, and 300 in DPB, to allow more complex networks to evolve. For each run of DPB, 25 instantiations of the cart-pole system were used to promote the learning of a general policy. The starting positions and velocities varied across these 25 runs. SAFS-NEAT used a window size of $w = 5$ generations, and set $\epsilon = 0.0005$.

4.5.2 Results and Discussion

Robot Auto Racing Simulator Figure 15 (a) compares the convergence rate of the three algorithms with 10 relevant and 10 irrelevant sensors. As a reference, it also shows the performance of NEAT with 10 relevant sensors and no irrelevant sensors. The negative performance impact of learning in the presence of irrelevant features is made clear by the difference in fitness between NEAT with zero irrelevant features, and NEAT with 10 irrelevant features. SAFS-NEAT is able to incrementally build a feature subset containing predominantly relevant features, enabling it to approach the performance of the reference curve; a performance that looks qualitatively good. FS-NEAT reaches an average performance that settles between the other two algorithms, scaling better than NEAT but worse than SAFS-NEAT. The standard deviation bars on the plot also show an interesting trend with respect to the feature selection algorithms. For FS-NEAT these error bars remain relatively large throughout learning. This is due to the random inclusion of irrelevant features in many selected subsets, which reduces the performance of the algorithm on average. Unlike FS-NEAT, SAFS-NEAT starts with large deviations, as different relevant features are selected early in learning, but these deviations shrink over time. This is due to the different runs converging on a similar set of relevant features, which in turn produce consistently fit networks.

As the number of irrelevant sensors increases from 0 to 80, similar trends noticed in plot (a) can be observed in plot (b) of Figure 15. For any number of irrelevant features tested, SAFS-NEAT consistently achieves a final performance value similar to the reference plot of NEAT with no irrelevant features. This shows that the feature selection mechanism employed by SAFS-NEAT can successfully select relevant features in high-dimensional environments. In contrast with this behavior, NEAT gets significantly worse as more irrelevant features are added to the problem because the evolutionary search has trouble making many good mutations simultaneously to improve the NNs. Only one or a few prosperous mutations may not be enough to drive the fitness of a NN high enough to guarantee its inclusion in the subsequent generation. Failure to survive to the next generation will cause these good mutations to die out from the population, preventing NEAT from progressing towards an optimal policy. The performance of FS-NEAT lies between SAFS-NEAT and NEAT for all settings of the irrelevant feature scaling experiments. It is able to scale better than NEAT by virtue of incorporating fewer features into its networks, requiring fewer simultaneous successful mutations to cause fitness improvement. It also clearly shows a sensitivity to the number of irrelevant features included in the problem, and attains increasingly lower fitness values as more irrelevant features are present in the problem. This is due to the high probability of randomly selecting an irrelevant feature into the network, forcing later generations to evolve around these errant features. SAFS-NEAT's sample-based feature selection strategy removes this dependency on the number of irrelevant features as they are all ranked low and hence not selected into the feature subset, regardless of their prevalence in the environment.

In these experiments there is a strong relationship between the aggregate number of samples observed by an agent, an evolutionary generation, and the fitness of the population. Specifically, the aggregate samples seen by an algorithm is given by $\sum_{i=1}^{200}\sum_{j=1}^{100} eval(NN_{ij})$, where $eval(\cdot)$ is a function that returns the number of samples observed while evaluating a network in the environment. Since each evaluation is restricted to 3000 time steps in this domain, there is a hard bound on the number of samples which can be seen overall, and in each generation. The actual number of samples used in any one generation can significantly vary below this bound, and is correlated with fitness of each member of the population. Low fitness generally costs few samples since the car crashes after very few time steps, while obtaining a high fitness score requires the use of all 3000 allotted time steps. Fitness differences between algorithms at any generation do not arise from an unfair advantage in number of samples available, but rather how effectively the evolution made use of all samples up to that generation.

Figure 16 gives empirical evidence to support this reasoning by presenting the total number of samples seen by each learning algorithm during the entire span of evolution. This plot follows the aggregate number of observed samples for each algorithm as the number of irrelevant features scales from 0 to 80, while holding the number of relevant features fixed at 10. We can see that both feature selection algorithms tend to interact with the environment a similar number of times across all settings. This number of interactions also corresponds with the results of the NEAT algorithm when no irrelevant features are present, meaning that performing feature selection does not introduce an additional burden on interacting with the environment. From Figure 15 we see that SAFS-NEAT evolves much more fit networks than FS-NEAT, especially as more irrelevant features are present in the environment. From this we can conclude that the feature relevance measure used by SAFS-NEAT makes more effective use of samples than evolutionary search of FS-NEAT when irrelevant features are present. Note that the decreasing trend of samples observed by NEAT naturally relates to the fitness performance of the algorithm. Less fit policies cannot observe comparatively many samples because they crash prior to using all 3000 time steps.

Figure 17 shows the size and quality of the selected feature subsets for the two feature selection algorithms, SAFS-NEAT and FS-NEAT, on Fiorano. Baseline NEAT is not shown because it

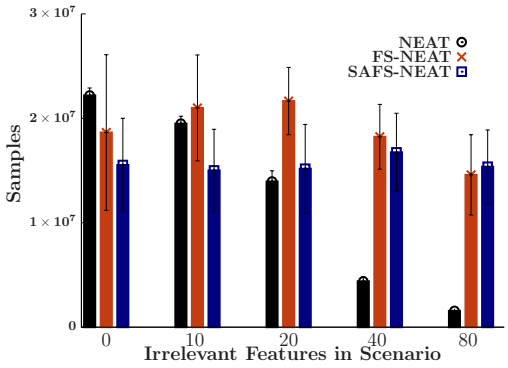


Figure 16: Average number of samples observed during 200 generations for each of the three algorithms on Fiorano as the number of irrelevant features increases from 0 to 80. Error bars denote the standard deviation of the 25 runs at each point.

always includes all features available in the environment. In Figure 17 the effects of increasing the number of irrelevant sensors in the environment can be seen. SAFS-NEAT tends to select around 9 features in total (open bars), since a good policy can usually be learned on a single track without all 10 relevant features. Out of these features, SAFS-NEAT selects only relevant features (solid bars) nearly all of the time. This is in sharp contrast to FS-NEAT, which selects increasing numbers of irrelevant features as it struggles to include relevant features. It ultimately selects around 5 relevant features on average, and the relevant fraction of the selected subset is very low when many irrelevant features are present in the environment. This contributes to the reduced learning performance seen in Figure 15(b).

Due to space limitations, results on the redundant scaling experiments are not reported in this work, however, we would like to note some general trends from their outcome. SAFS-NEAT continued to perform near the reference curve in all scenarios, and FS-NEAT performed similarly well. NEAT did suffer slightly as many features were included due to the size of the networks, but not nearly as much as in the irrelevant scaling experiments. FS-NEAT was able to perform similarly well as SAFS-NEAT because with only relevant features to choose from, adding a feature would not tend to harm the policy much, allowing FS-NEAT to find and maintain a high-fitness policy.

Double Inverted Pendulum Balancing Figure 18 shows the average fitness performance of the three algorithms as the number of irrelevant features increases in the double inverted pendulum balance (DPB) environment. Fitness indicates the number of time steps that both pendulums remained balanced on the cart, with 100,000 being the maximum possible in our setup. Similar to the above results, the performance of NEAT on the scenario with no irrelevant features is shown as a reference. We see in Figure 18(a) that 6 irrelevant features causes a large performance decrease for NEAT, and that NEAT makes nearly no progress towards π^* when there are at least 12 irrelevant features in the environment. FS-NEAT is better able to cope with irrelevant features, and achieves around 90% of the maximum fitness value when 6 or 12 irrelevant features are present as evidenced in Figure 18(b). Its performance drops off sharply when 24 or 48 irrelevant features are in the environment. On the other hand, SAFS-NEAT is able to learn a near optimal policy in all four scenarios, and even slightly outperforms the reference curve on all settings.

This at first may seem counter-intuitive, but after examining the results of SAFS-NEAT we

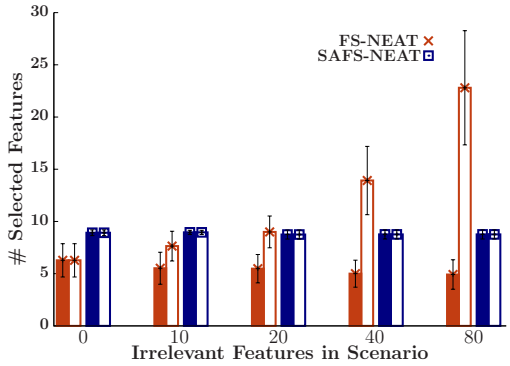
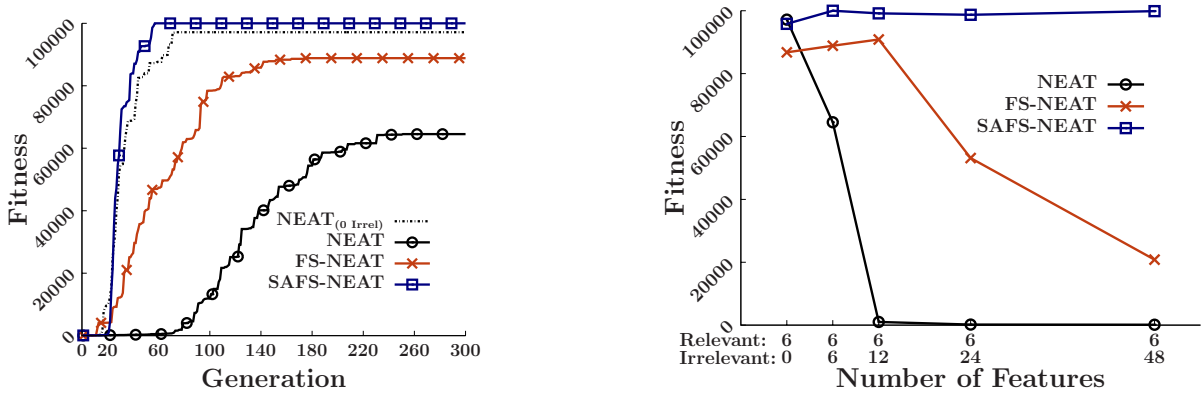


Figure 17: Average number of selected features in total (empty) and relevant (solid) at generation 200 on Fiorano as the number of irrelevant features increases from 0 to 80. Error bars denote one standard deviation of the 25 runs at each point.



(a) 6 Irrelevant Features

(b) Irrelevant Scaling

Figure 18: (a) Fitness at each generation during policy learning for NEAT, FS-NEAT, and SAFS-NEAT with 6 irrelevant features. (b) Shows the fitness of each algorithm at generation 300 as irrelevant features increase from 0 to 48.

observed that many NNs connected the inputs to only specific output nodes. NEAT networks always begin fully connected, and so have a larger number of connections which may be mutated, increasing the policy search space and reducing the chances of evolving π^* in the duration of the experiment. Unlike the RARS environment, SAFS-NEAT shows very little improvement in the starting generations on the double pendulum balance. Due to feature interactions in this domain, several features must first be included in the subset before learning progress can be made, as evidenced by the flat fitness curve for approximately 20 generations. Once an appropriate feature set is determined, SAFS-NEAT rapidly achieves a near optimal fitness. Standard deviation error bars are not shown for improved clarity of these plots. FS-NEAT in this environment magnifies its large standard deviation trend observed in the RARS problem, obscuring the results. SAFS-NEAT and NEAT both have small deviations in this problem after around 100 generations as all runs of these algorithms begin to converge to their final fitness values.

Space limitations prevent us from including sample usage results for DPB, but overall trends observed from the RARS domain are repeated for DPB. Feature selection does not increase the

sample requirements of the NEAT algorithm, and given the difference in performance, SAFS-NEAT can be said to make more efficient use of the samples than FS-NEAT. SAFS-NEAT is also able to select a near purely relevant feature subset throughout the different scenarios, selecting around 5 features on average. FS-NEAT again includes many irrelevant features in the high-dimensional settings, causing evolution to be unable to find an effective policy in those scenarios.

The results presented in Sections 4.5.2 & 4.5.2 have served to illustrate several key points: (i) irrelevant features can prohibit the learning of a good policy; (ii) the existence of redundant information can sometimes degrade policy learning without feature selection; (iii) SAFS-NEAT is able to effectively eliminate irrelevant and many redundant features, leading to good selected feature subsets and fit policies; (iv) SAFS-NEAT is more effective than FS-NEAT, especially for an environment with many irrelevant features but few relevant features. From these points we can conclude that SAFS-NEAT is a sample efficient feature selection algorithm for RL that improves the scalability of the NEAT algorithm.

5 CONCLUSIONS

In this effort, we have developed approaches for performing machine learning in complex environments that fall within two major areas: state abstraction and feature selection techniques. Both areas show promise for future work and have applicability to real-world Air Force problems with high complexity.

Specifically, we've detailed the following:

An embedded feature selection algorithm which incorporates a sequential forward search into the neuroevolutionary function approximation method NEAT for reinforcement learning. Our results demonstrate the effectiveness of IFSE-NEAT at identifying relevant features and eliminating irrelevant ones. This ability enables IFSE-NEAT to converge upon higher quality policies using simpler networks in fewer generations than either NEAT or FS-NEAT. However, although IFSE-NEAT more efficient than wrapper methods, the incremental search for relevant features adds significant computational cost when compared to the other NEAT variants. Possible future directions include investigating the parallelization of the algorithm to help mitigate this cost, and further study on the generalization ability of the simple NN solutions found by IFSE-NEAT.

We have presented the Maximum Distance Separation algorithm which seeks to automatically partition a state space based on dense regions of observations. This method has been shown to improve the learning rate as compared to using a fixed abstraction or naively altering the number of states during evolution on a challenging reinforcement learning problem, and they allow the learner to consider only a small number of states while doing so. This work yields itself to several promising future directions. As illustrated by the mountain car results, intelligent aggregation is not always beneficial. Identifying these problems during the abstraction generation procedure could be one future area of interest. There is no need to limit MDS to the one dimensional state space used by RL-SANE, and applying this method to the ground state space or in combination with a different dimensionality reduction technique may prove useful. In general, the interplay between state abstraction and dimensionality reduction could be an interesting avenue of future research.

We have presented three types of automatic state abstraction techniques, mutation methods that make use of ANNs to abstract the space, Maximum Distance Separation which seeks to partition a space based on dense regions of observations, and Temporal Relative Extrema which builds abstract states by separating observations that lead to previously seen areas of the state space. Each of these

methods has been shown to improve the learning rate as compared to using a fixed abstraction, and they make use of only a small number of states while doing so. One future direction of this work is to experiment with the techniques on higher dimensional state spaces, instead of restricting them to the one dimensional abstract space at work in the RL-SANE algorithm. Another interesting possibility is to relax the current abstraction conceptualization and not require that a partition of the space be determined; instead only focus on those areas that need increased resolution.

In this work, we have proposed a novel feature selection algorithm for RL with continuous action spaces, SAFS-NEAT, that automatically and efficiently discovers a good set of features for learning a generalized policy. To achieve this, it embeds a sequential forward search procedure into the NEAT policy search algorithm, and applies an efficient method to evaluate the goodness of features via previously collected samples. We have demonstrated its ability to handle high-dimensional state spaces in two challenging problems, where it outperformed a feature selection algorithm that did not exploit sample knowledge. There are several directions for possible future work. One is to investigate filter feature selection algorithms that could be used in place of mutual information. Another would be to investigate feature selection approaches embedded into other learning algorithms. We believe that further investigating the usage of samples in feature selection can lead to more efficient learning algorithms which are better able to scale to real-world high-dimensional RL problems.

EvoTC and CMAC were able to solve both benchmark problems it does not appear that either method will scale adequately as the number of features that describe the state space is increased. Both methods are tile coding based and are linear abstractions of the state space. As a result, although the abstract state spaces found by these methods are significantly smaller than the actual state space, they will still scale proportionally as the number of features is increased. It may be the case that non-linear state abstraction methods such as RL-SANE [10] are necessary as the number of features are increased. Real world applications have large continuous state spaces that prevent the use of RL algorithms. State abstraction methods such as tile coding are necessary in order to apply RL to non-trivial problems. Fixed tile coding algorithms such as CMAC can be effective as long as the tiling scheme is properly designed. Adaptive tile coding methods like ATC and EvoTC are appealing because they do not require manual design of the state abstraction. In this paper we introduced EvoTC and showed how it is able to abstract the state space more effectively than CMAC and ATC on two continuous state space problems. Not only was EvoTC able to outperform CMAC and ATC in terms of abstraction power it was able to reduce the classical mountain car domain to a problem consisting of just two states. This result highlights the power and importance of automated state abstraction methods. Although EvoTC was able to very effectively abstract the state space of the mountain car problem it does not appear that the approach will scale well as the number of features that describe the domain are increased. We believe this is due to the linear nature of the tiling abstraction. Although the tilings are gross abstractions of the state space the dimensionality of the abstract state space is the same as the original state space. In future work we will explore the derivation of non-linear state abstraction devices such as multi-layered feed forward neural networks [10] and examine how they scale. Non-linear state abstractions may be able to find more efficient abstractions of mutli-dimensional state spaces enabling them to scale more effectively as the number of features is increased.

6 REFERENCES

References

[1] J. S. Albus, "A theory of cerebellar functions," *Mathematical Biosciences*, vol. 10, pp. 25–61, 1971.

[2] W. T. B. Uther and M. M. Veloso, "Tree based discretization for continuous state space reinforcement learning," in *AAAI '98/IAAI '98: Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1998, pp. 769–774.

[3] S. Whiteson, M. E. Taylor, and P. Stone, "Adaptive tile coding for value function approximation," University of Texas at Austin, Tech. Rep., 2007.

[4] I. Miller, W.T., F. Glanz, and I. Kraft, L.G., "Cmas: an associative neural network alternative to backpropagation," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1561 –1567, oct 1990.

[5] R. S. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," in *NIPS*, 1995, pp. 1038–1044.

[6] F. Gomez, J. Schmidhuber, and R. Miikkulainen, "Efficient non-linear control through neuroevolution," in *Proceedings of the European Conference on Machine Learning*, 2006, pp. 654–662.

[7] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992.

[8] L. Li, T. J. Walsh, and M. L. Littman, "Towards a unified theory of state abstraction for MDPs," in *In Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics*, 2006, pp. 531–539.

[9] A. K. Mccallum, "Reinforcement learning with selective perception and hidden state," Ph.D. dissertation, The University of Rochester, 1996, supervisor-Ballard, Dana.

[10] R. Wright and N. Gemelli, "State aggregation for reinforcement learning using neuroevolution," in *ICAART 2009 - Proceedings of the International Conference on Agents and Artificial Intelligence*, 2009, pp. 45–52.

[11] K. O. Stanley and R. Miikkulainen, "Efficient reinforcement learning through evolving neural network topologies," in *Proceedings of GECCO*, 2002, pp. 569–577.

[12] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[13] H.-P. Kriegel, P. Kröger, and A. Zimek, "Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering," *ACM Trans. Knowl. Discov. Data*, vol. 3, no. 1, pp. 1–58, 2009.

[14] D. Comaniciu and P. Meer, "Mean shift: a robust approach toward feature space analysis," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, pp. 603–619, 2002.

[15] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[16] H. Liu and L. Yu, "Toward integrating feature selection algorithms for classification and clustering," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 4, pp. 491–502, 2005.

[17] J. Z. Kolter and A. Y. Ng, "Regularization and feature selection in least-squares temporal difference learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*, 2009, pp. 521–528.

[18] R. Parr, L. Li, G. Taylor, C. Painter-Wakefield, and M. L. Littman, "An analysis of linear models, linear value-function approximation, and feature selection for reinforcement learning," in *Proceedings of the 25th International Conference on Machine learning*, 2008, pp. 752–759.

[19] M. Kroon and S. Whiteson, "Automatic feature selection for model-based reinforcement learning in factored mdps," in *Proceedings of the 2009 International Conference on Machine Learning and Applications*, ser. ICMLA '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 324–330.

[20] S. Whiteson, P. Stone, and K. O. Stanley, "Automatic feature selection in neuroevolution," in *Proceedings of GECCO*, 2005, pp. 1225–1232.

[21] F. J. Gomez and R. Miikkulainen, "Solving non-markovian control tasks with neuroevolution," in *In Proceedings of the 16th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1999, pp. 1356–1361.

[22] R. Sutton and A. Barto, *Reinforcement learning: an introduction*. MIT Press, 1998.

[23] L. P. Kaelbling, M. Littman, and A. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[24] H. Hachiya and M. Sugiyama, "Feature selection for reinforcement learning: Evaluating implicit state-reward dependency via conditional mutual information," in *Proceedings of the ECML*, 2010, pp. 474–489.

[25] A. Nouri and M. Littman, "Dimension reduction and its application to model-based exploration in continuous spaces," *Machine Learning*, vol. 81, pp. 85–98, 2010.

[26] S. Whiteson and P. Stone, "Evolutionary function approximation for reinforcement learning," *Journal of Machine Learning Research*, vol. 7, pp. 877–917, May 2006.

[27] M. Tan, M. Hartley, M. Bister, and R. Deklerck, "Automated feature selection in neuroevolution," *Evolutionary Intelligence*, vol. 1, no. 4, pp. 271–292, 2009.

[28] R. Wright, S. Loscalzo, and L. Yu, "Embedded incremental feature selection for reinforcement learning," in *ICAART 2011 - Proceedings of the 3rd International Conference on Agents and Artificial Intelligence, Volume 1 - Artificial Intelligence, Rome, Italy, January 28-30, 2011*, 2011, pp. 263–268.

[29] P. Devijver and J. Kittler, *Pattern Recognition: A Statistical Approach*. Prentice Hall International, 1982.

[30] A. Castelletti, S. Galelli, M. Restelli, and R. Soncini-Sessa, "Tree-based variable selection for dimensionality reduction of large-scale control systems," in *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2011 IEEE Symposium on.* IEEE, April 2011, pp. 62–69.

[31] H. Liu and H. Motoda, Eds., *Feature Extraction, Construction and Selection: A Data Mining Perspective.* Boston: Kluwer Academic Publishers, 1998, 2nd Printing, 2001.

[32] H. Peng, F. Long, and C. Ding, "Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 27, pp. 1226–1238, 2005.

[33] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike adaptive elements that can solve difficult learning control problems," *Artificial neural networks: concept learning*, pp. 81–93, 1990.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ANJI	Another NEAT Java Implementation
ANN	Artificial Neural Network
ATC	Adaptive Tile Coding
CMAC	Cerebellar Model Articulation Controller
DPB	Double Pole Balance
EvoTC	Evolutionary Tile Coding
FS-NEAT	Feature Selective NEAT
IFSE-NEAT	Incremental Feature Selection Embedded in NEAT
MDP	Markov Decision Process
MDS	Maximum Density Separation
NEAT	NeuroEvolution of Augmenting Topologies
NN	Neural Network
RARS	Robot Auto Racing Simulator
RL	Reinforcement Learning
RL-SANE	Reinforcement Learning using State Abstraction via NeuroEvolution
SAFS-NEAT	Sample Aware Feature Selection embedded in NEAT
SARSA	State Action Reward State Action
SFS	Sequential Feature Selection
TRE	Temporal Relative Extrema